1985 Mid-Year Report
NASA Grant NSG 1471

THE EMBEDDED OPERATING SYSTEM PROJECT

Principal Investigator
Roy H. Campbell

Research Assistants
Raymond B. Essick
Judith Grass
Dirk Grunwald
Pankaj Jalote
Kevin Kenny
David A. McNabb

Software Systems Research Group
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 West Springfield Avenue
Urbana, Illinois 61801-2987
(217) 333-0215

NASA Grant NSG 1471

The Embedded Operating System Project

Mid-Year Report, May 1985

Principal Investigator
Roy H. Campbell

Research Assistants
Raymond B. Essick
Judith Grass
Dirk Grunwald
Pankaj Jalote
Kevin Kenny
David A. McNabb

Software Systems Research Group
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 West Springfield Avenue
Urbana, Illinois 61801-2987
(217) 333-0215

I

ABSTRACT

This progress report describes research towards the design and construction of embedded operating systems for real-time advanced aerospace applications. The applications concerned require reliable operating system support that must accommodate networks of computers. The report addresses problems that arise in the construction of such operating systems, reconfiguration, consistency and recovery in a distributed system, and the issues of real-time processing. We include a thesis that provides theoretical foundations for the use of atomic actions to support fault tolerance and data consistency in real-time object-based systems. In particular, this report addresses:

- Atomic Actions and Fault-Tolerance Issues
- Operating System Structure
- Program Development
- A Reliable Compiler for Path Pascal
- Mediators: A Mechanism for Scheduling Distributed System Processes

This document reports the status of various experiments designed and conducted to investigate embedded operating system design issues. To support EOS, our experimental real-time Embedded Operating System design, we are constructing a portable object-based development system called INDEED. INDEED provides an incremental development environment aimed at the particular needs of object-based real-time system construction. EOS is representative of a family of operating system designs based on a General Layered Operating System construction methodology called GLOSS. In addition, we have implemented a portable and reliable compiler for Distributed Path Pascal, the real-time programming language in which we propose to conduct many of the experiments. This compiler is in production use at more than ten international sites.

Using the real-time programming techniques developed in co-operation with NASA in earlier research, the project staff is building a set of modular components for a family of real-time distributed operating systems as part of an experiment to evaluate the proposed object-oriented approach.

Key components of the design are being programmed in Distributed Path Pascal, a high-level programming language that incorporates strong-typing, allows object-oriented programming, modularization of code, separate compilation, and fast real-time execution. The real-time performance of these components is being studied by simulation and by experimental evaluation on stand-alone machines. The research includes investigation of distributed kernels, scheduling, management and naming of distributed resources, exception handling, fault-tolerance, and an incremental development environment for distributed software.

1. Project EOS Overview

Since 1979, the Software Systems Research Group at the University of Illinois has been working with Dr. Edwin C. Foudriat of NASA Langley to develop methods and techniques for the construction of real-time embedded operating systems for aerospace applications. Many research contributions in real-time scheduling, fault-tolerant software, operating system structure, object-oriented systems programming, networking and distributed computing have been produced by this co-operative effort. The major practical research contribution produced is an experimental real-time programming and simulation language called Distributed Path Pascal [17]. Distributed Path Pascal incorporates strong-typing and allows object-oriented programming, modularization of code, separate compilation, and fast real-time execution.

Distributed Path Pascal has been the development vehicle used to study many prototype systems and research issues. The group has designed several small operating system components [8,9,14,17,22,27,29,36,40,41,42,43,44,46] based on an object-oriented view of a computer system. This view accommodates the design of autonomous operating system components networked together as "remote objects" [17,27]. Backward error recovery, conversations, the deadline mechanism [15,32,36] and recoverable objects [31] have been prototyped using Path Pascal. The research project has also produced major contributions documented in the 20 published papers, 7 M.S theses, 5 Ph.D. theses and some 29 technical reports listed in Appendix A. These contributions covered many aspects of system design including protection [22,30,40,41,43] fault tolerance [32,36,39,45,46], fault tolerance methodology [1,3,4,6,20,31] fault-tolerance in real-

time systems [3,18,28,32], atomicity, fault-tolerance, and consistency [1,2,48], and distributed data base consistency [21,50].

Our current research concentrates on applying the results of our previous research to the design and construction of components of a prototype distributed real-time embedded operating system (EOS). The major requirements for EOS are listed below.

Real-Time Response. Components and subsystems of the application must have support to enable them to respond to I/O events in real-time; that is, fast enough to provide control for the physical system in which the computer is embedded.

Reliable Operation and Fault Tolerance. System components may be used to implement critical life-support and hardware survival functions, and must have a very low likelyhood of failure. Fault tolerant techniques should be employed to achieve levels of reliability beyond those that can be achieved by conventional software engineering methodologies.

Autonomous Operation. The system should be a dynamically reconfigurable collection of distributed, loosely-coupled, highly autonomous components. Such systems support failure isolation, standby sparing, triple-modular redundancy, and majority voting. The modularization of components improves reliability and facilitates maintenance.

Design and Maintenance Support. The development of an application will consist of the design, construction, configuration, testing, and maintenance of highly autonomous objects and collections of objects. This development process must be supported by

appropriate tools and facilities. In particular, these tools must allow fast prototyping, system instrumentation and debugging mechanisms, dynamic upgrading of object implementations, reconfiguration, reusable software components, test-bed validation, performance evaluation and tuning.

This report describes the results of the EOS project for the six months from November 15th through the present. During this time we have:

- developed programming and fault-tolerant system concepts based on atomic actions;
- designed a development environment for object-oriented systems which aids in the construction of distributed software;
- investigated mediators, a new modular scheduling scheme for real-time systems, and language primitives to implement that scheme in distributed systems;
- enhanced and distributed a portable Path Pascal code-generating compiler for UNIX[1] and stand-alone systems which will support production development of Path Pascal programs;
- continued the design of the overall structure of EOS.

In section 2 of this report we describe our work on fault tolerance in distributed systems through atomic actions.

Section 3 contains a description of INDEED, an INcremental Development Environment for Extensible Distributed systems. INDEED provides a means to prototype, extend, debug, instrument, test and maintain object-based embedded systems. The system supports dynamic reconfiguration, remote operations, and distribution on a network of processors.

[1] UNIX is a trademark of Bell Laboratories.

Section 4 discusses the issues relating to the structure of operating systems. The General Layered Operating System Structure, GLOSS, provides a methodology for building a family of operating systems from reusable components. Depending upon the choice and manner in which these components are combined, systems with different properties can be obtained.

Section 5 contains a proposal to solve several language design issues related to the problem of specifying the scheduling of reliable real-time systems. Potentially, the scheme may be extendible to support fault tolerance and atomic actions. The proposed solutions are described in the form of system and language primitives for supporting such facilities.

In Section 6 we describe the current status of Distributed Path Pascal. A new production Distributed Path Pascal compiler has been completed in which the code generation phase is the same as that used to support the Berkeley version of the UNIX portable C compiler. The front end of this compiler is implemented using an LALR parser and components taken from the Berkeley Pascal compiler. In addition to offering improved performance, the software facilitates porting Path Pascal to many different machines, allowing a reliable production environment for both stand-alone and UNIX-based Distributed Path Pascal. Implementations now exist on both the VAX and MC68000, and under both Berkeley 4.2 UNIX and XENIX[2].

[2] XENIX is a trademark of Microsoft.

2. Fault Tolerance

The last six months has seen the completion of a study of the nature of fault-tolerant provisions in asynchronous systems. Several papers have been completed and accepted for publication in *Transactions of Software Engineering* documenting our research [1,2,3,4]. A Ph.D. thesis was completed [Appendix D] providing a summary of much of this research. The research has produced:

- a framework for exception handling in asynchronous computer systems;
- a practical implementation of the exception handling scheme in a message based system of concurrent processes using a CSP-based notation;
- demonstrations of using the exception handling schemes to support backward and forward error recovery;
- arguments supporting the thesis that atomic actions are fundamental to the provision of fault tolerance.

Future directions for fault-tolerant software studies in Project EOS will address practical issues of providing the fault-tolerant schemes proposed within an embedded real-time operating system.

3. INDEED

The design and construction of embedded real-time operating systems are formidable tasks often involving large groups of programmers and expensive resources. While we have shown that the use of a high-level, strongly-typed, object-based concurrent programming language greatly facilitates these efforts, many difficulties remain. We are designing a portable incremental program development environment for object-based systems which will provide the additional support needed during construction of

advanced real-time embedded systems.

Recent research has clarified some of these problems and identified additional requirements for our development environment. Among these are:

- ease of rapid prototyping,
- immediate testing of new system components (objects) within running systems,
- provision for object stubs,
- mechanisms to replace stubs with actual implementations while minimizing perturbations of the system,
- flexible and dynamic instrumentation of system objects,
- preservation of the maximum possible portion of consistent system state during incremental development,
- dynamic replacement of system objects by newer versions,
- and dynamic reconfiguration of the overall system structure.

We believe that each of these requirements can be achieved within the framework of the existing Path Pascal language definition through development of a parallel, extended semantics for identifier scopes and object namespaces.

During the last six months we have investigated details of the design of INDEED and studied the relative difficulties of implementing INDEED using one of our existing Path Pascal compiler systems. We are preparing a document detailing the required modifications, comparing existing compilers for feasibility of these modifications, and suggesting development stages and milestones. Some consideration of the feasibility of an INDEED-like environment for other modern concurrent programming languages

may also be included in this report.

Work has already begun on an interpreted INDEED prototype based on the tree-based intermediate code, Tcode[3] Two existing Tcode compilers (one for the Series 1[4] and one for an interpreter [25]) and several code generators[5] have been created. Interpreted Tcode provides an excellent vehicle to develop INDEED algorithms and concepts further and is likely to be enhanced in the near future.

While some of these development environment requirements might also be useful capabilities of the systems produced, the overhead of the dynamic reconfiguration features could be too costly in light of the stringent requirements of the final real-time systems. However, these expanded capabilities are implemented in INDEED as orthogonal and optional constructs. If these features are used for development purposes only, then the final production system can be generated by a standard Path Pascal compiler and thus avoid all such overhead.

The INDEED system can also be used to configure systems of objects into arbitrary graph structures, permitting hybrid arrangements which have no corresponding valid Path Pascal equivalent. Experiments on such systems are not currently planned. Similarly, objects can be arranged into strict hierarchies of layers, with each layer extending, refining, restricting, or simply transmitting operations of the parent layers. Such approaches are described later in this report under the section on GLOSS. We

---

[3] Beshers, G. M., D. A. McNabb, and R. H. Campbell, "TCODE: An Intermediate Code for the Path Pascal Compiler," Technical Report UIUCDCS-R-81-1060, Department of Computer Science, University of Illinois, 1981.

[4] G. Beshers, "Tcode Compiler for the Series-I," Final Report: IBM Series 1 Project, 1983.

[5] D. McNabb, Internal Report.

expect that INDEED will provide an excellent test-bed for experiments with GLOSS designs, and will facilitate testing of families of GLOSS systems.

4. GLOSS

Operating systems can provide many useful support facilities for programming in high-level abstract languages. The implementation of such facilities may be complex. However, real-time critical processes require fast, economical support from an operating system. One traditional approach is to provide support for real-time and high-level functionality within an integrated operating system. These operating systems can become large and complex, are often not modular in design, and require specialized primitives which differ for real-time and high-level processes. Using customized operating systems on independent but communicating systems is one method for providing both types of support.

A desirable goal is to reduce the complexity of real-time operating systems that provide high-level functionality. Less complex operating systems are more easily developed and maintained. Another goal is to provide a consistent interface between the process and the operating system regardless of whether real-time or high-level behavior is desired. Consistent interfaces between the process and the operating system in both environments enable the programmer to rapidly and easily change environments as desired. Distributed applications should be run in a high level environment offering, for example, distributed filesystems, while real-time processes are run in an environment offering minimal operating system overhead.

The GLOSS concept is intended to facilitate the design of a single family of operating systems to support both real-time and high-level environments. However, unlike other operating system designs, the approach allows a separation of real-time concerns from functionality and permits the exploitation of multiprocessors in the support of real-time performance. Real-time functions can suffer in environments with complex operating systems and the accompanying overhead. However, high-level programming language functions can benefit greatly from the higher degree of support provided by an operating system. If software developers are to provide environments for real-time and high-level functions, it is convenient to have a similar, if not identical, set of system services in both environments. GLOSS permits the development of such tailored, but generic, environments.

The GLOSS methodology provides a structuring mechanism for operating systems allowing real-time functions to run with minimal operating system overhead. High-level functions can support features such as triple-modular redundancy, atomic actions, stable storage and distributed filesystem access.

4.1. Layering

Layering an operating system provides one mechanism for offering both real-time and high-level performance within a single operating system. The high-level functions such as distributed filesystems are implemented in separate layers. Processes are given access to a particular set of layers. High performance processes have direct access to the inner layers of the kernel while processes desiring the more complex functions access the outer layers of the kernel.

4.2. Consistent Layer Interfaces

By choosing to have identical interfaces between each layer of the operating system, we gain the ability to easily stack layers to achieve a desired level of functionality. Each process is given a set of layers most appropriate to its task. This appropriate set of layers is determined at execution time and can change with each invocation of the program. It is possible for concurrent instances of a program to use different layers of the operating system. Programs need not be recompiled to use specific layers; they are merely started with an appropriate set of underlying layers.

4.3. Related Concepts

The concept of consistent interfaces is not novel. The idea of device independence is similar to the GLOSS concept in many ways. Device independence is usually associated with I/O operations. The set of operations on a GLOSS layer would include the entire set of "system calls", or "service traps".

Another related scheme is the UNIX "STREAM" or "stackable line discipline".[6] Streams consist of a series of processing modules whose only interface is a set of read and write routines. These modules can be assembled in any order to provide a variety of functions that were previously implemented by adding special cases to a single large body of code.

[6] The Ritchie paper in the 1984 UNIX issue of the Bell Technical Journal.

4.3.1. Device Independence

Device independence is one example of a consistent interface. Prior to its introduction, programs contained specialized and separate code to output to disks, printers, terminals, and other programs. In an operating system designed with device independence, a program need not concern itself with whether the output is going to a printer, disk file, or another program. It continues to use the same set of I/O primitives to generate output.

4.3.2. UNIX Streams

The UNIX teletype driver has been continually modified until it is now a important and complex piece of code. The reason for this is that the teletype driver has born the brunt of additions to support such features as differing "line disciplines", erase and kill handling, and other somewhat related tasks. In an effort to reduce the complexity of the teletype driver and simplify the structure of related objects in the UNIX kernel, Dennis Ritchie developed the STREAMS concept.

The STREAM concept includes a set of processing modules, each acting as a filter, sitting between a user process and a device (or another user process). These modules have consistent interfaces and hide the type of module with which communication occurs. The processing modules implement the functions that include erase and kill processing, front-end communications protocols for networked or virtual terminals, and more complex tasks. Additional processing modules can be dynamically pushed onto a given set of modules; thus the name "stackable line discipline".

4.4. Current Status

In this section, we detail progress made in the construction of GLOSS-based systems. There are currently two primary efforts in the prototyping and development of GLOSS systems. The UNIX United effort is concerned with prototyping the GLOSS concept in the UNIX kernel. The Path Pascal effort is concerned with developing an implementation of GLOSS.

The Path Pascal GLOSS effort is concerned with modifying the language and language run-time systems to support interchangeability and stackability of service routines. We are examining mechanisms for replacing the set of Pascal "standard procedures" with references to an object. We will then permit such objects to be replaced by another of our own choosing on a per-process basis. These objects could be linked in chains; each object in the chain representing a GLOSS layer and presenting the same interface to its clients.

4.4.1. UNIX System V

The 3b2 UNIX United implementation is using a prototype of GLOSS to implement a distributed UNIX system. The System V UNIX kernel is augmented with a device driver that traps user process service requests to a UNIX United layer also implemented within the kernel as a device driver. The UNIX United layer determines the appropriate interface to use to service a given system request and contains the linkages to allow the execution of the service to occur.

Vince Russo is in the process of moving the functions of the UNIX United "layer" into a kernel layer using AT&T financial support.[7] The objectives of this port include producing a distributed UNIX and implementing it in such a fashion that a minimal number of modifications have to be made to the original UNIX kernel.

To minimize the changes required to the UNIX kernel, UNIX United is being implemented as a layer contained in a pseudo device driver. This driver can be optionally loaded; the determination coming at boot time. When the driver is loaded, the UNIX kernel calls a driver specific routine to initialize the device. In UNIX United, we overlay the normal system call entry point table with a new table of services. When user processes trap into the UNIX kernel, they are redirected through this table to the routines specific to the layer instead of the normal UNIX system call routines. The routines in this layer may then invoke the true UNIX system call routines via a saved copy of the original system call table.

This implementation is not strictly a GLOSS implementation since it redirects all processes through the same additional layer. Future work includes carrying per-process pointers around which describe the particular system call entry table to use when handling requests from the user process. This adds an extra level of indirection to the dereferencing of the system call but allows us to use a per-process value to determine the actual set of layers instead of a system wide progression of layers.

The actual layered GLOSS model is difficult to implement with the current UNIX base. In the original PDP-11 versions of UNIX, the kernel ran in an address space

[7] See also, Mark Tuomenoska, "Loadable Illinois Newcastle Connection," M.S. Thesis, 1985.

separate from the user's address space. Special routines had to be invoked to move data between the two address spaces. As UNIX has been ported to machines with larger address spaces and virtual memory, versions where the user and kernel share the same address space are becoming more common. However, the model for copying data between user and kernel address space continues to use special routines. In UNIX United, the GLOSS layers actually reside in the kernel's address space. On UNIX systems where the kernel and user share a single address space there is no problem other than circumventing the protection mechanism to prevent an inner layer of the kernel from declaring an address invalid when it refers to data within a layer instead of user space. On UNIX systems where the kernel and user are in separate address spaces, the internal layers must place data to be transfered to inner layers in the user address space. The 3b2 system is one example of this case. The UNIX kernel on the 3b2 runs using physical addresses and the user process uses virtual addresses.

This summer, the UNIX United kernel layer is functioning but is incomplete. Calls referencing remote systems are detected and diverted to the correct service routines on the local hosts. These service routines are able to communicate with the remote host, making requests for actions to be performed on the remote host. The resulting system is smaller, more portable, and faster than the original Newcastle Connection which implemented UNIX United.

4.4.2. Path Pascal

The effort of making a Path Pascal based GLOSS system will be based on the new Grunwald compiler generating native code for the VAX and 68000 processor families.

We are currently looking at the minimal number of necessary language extensions to implement GLOSS layering. We are also investigating the generic set of primitives which should be supported by a given layer. Related work on INDEED should help with determining the linkages between layers.

Our current plans are to treat calls to the set of "Standard Procedures" or CSP's as calls to a standard object. This object will be passed invisibly to each process.

Ways to define a "generic" object are being investigated. Generic objects will be used to parameterize the actual object a process will use when requesting services from the operating system.

Our current plans include adaptation of the "execute" and "service" statements [22] to support GLOSS as an extension of the Grunwald compiler. These statements would be used to operate on Path Pascal objects which encapsulate user programs. To a user program, a GLOSS layer will be represented as a "standard service object" and a number of these would be linked to form a chain to the most fundamental set of runtime support routines.

Further work will concentrate on the selection of appropriate set of interfaces to define layers that have the appropriate degree of encapsulation. An inappropriate set of routines in the interface can cause problems in the isolation of specific data. Another topic for investigation is the analysis of the actual flow of communication. Some activities will always require action at the innermost layer of the operating system. An approach where such actions can be sent directly to the innermost layer without any intervention by the intervening layers could provide improved performance for

processes operating in environments with multiple-layers between themselves and the innermost kernel. An implementation scheme based on the device driver UNIX United approach may provide an efficient mechanism for supporting multiple layers and will be investigated.

5. Mediators

Since programming a real-time Embedded Operating System requires a good deal of programming of synchronization and scheduling, a portion of our research has been devoted to language design and programming methods for concurrent programming. Our interest in the topic is prompted by the observation that many existing tools for concurrent programming overly constrain concurrency, complicate scheduling and do not allow a modular approach to the specification of timing constraints.

The *mediator* construct is proposed as an answer to these problems. Appendix E describes the design of the mediator construct. During the past year we have been experimenting with this design and applying it to a number of familiar and unfamiliar synchronization and scheduling problems. We believe that the design is flexible enough to adapt to a wide range of problems and to facilitate good modular programming design.

Within the next year we plan to develop a more formal specification of the semantics of the mediator construct. We hope that this may serve as a check on the design, and as a useful tool for implementation. Ultimately, this will serve as the basis for a proof system.

Implementation of mediators is a long term goal. Although many of the basic algorithms required are well known, implementation can not proceed before the design is complete. The challenge in implementation is less one of discovering suitable algorithms as in combining familiar ones into a new configuration.

6. Native Path Pascal Compiler

A native code producing portable Path Pascal compiler for the UNIX environment has been completed and tested [Grunwald,85]. An M.S. thesis documents the design of the compiler and is included as Appendix C. Various manuals document the run-time system, the compiler and the intermediate code used. A new Path Pascal manual is included as Appendix F. The new Path Pascal compiler supports:

- separate compilation of procedures, processes, and objects;
- interrupt processing via "DOIO" which is mapped, in UNIX implementations , to the UNIX signal mechanism;
- dynamic storage allocation for processes;
- invocation of Fortran 77, C, Prolog and assembler routines from Path Pascal;
- a *finally* statement which is executed automatically, in a manner similar to that of the *initially* statement, just before an object's storage is released by a dispose statement or by exit from a block;
- code generation for VAX and MC68000 computers;
- a run-time support system for 4.2 Berkeley UNIX;
- a run-time support system for XENIX.

The initial version of the Path Pascal compiler was created for the VAX environment running Berkeley 4.2, and then ported to the SUN 68000 Berkeley 4.2 system. Under the SUN 4.2 system, with 2 megabytes of main memory, Path Pascal can support the concurrent execution of over four thousand processes. Symbolic debugging is supported through the use of standard UNIX utilities. Recently, the compiler was ported to the IBM S9000, a 68000-based system running XENIX. The rapid development of Path Pascal compilers for these three systems clearly demonstrates the portability of the new compiler. A partial list of sites to which we have distributed Path Pascal is included in Appendix G. An example of a Path Pascal program showing the syntax for separate compilation is shown in Appendix H.

7. Conclusion

Project EOS has made progress towards the goal of producing a design for a reliable real-time operating system for embedded systems. Several of the goals that we proposed to accomplish we were unable to pursue because of limited funds. However, the project has continued to produce contributions towards an understanding of fault-tolerant software, distributed systems, layered operating systems, and scheduling provisions within a real-time system. The new Path Pascal compiler is reliable and produces efficient code that can be used to build operating system prototypes. The compiler is already in widespread international use. This Fall, we hope to make progress in developing EOS operating system components including a distributed kernel.

APPENDIX A

Project EOS Papers, Theses, and Technical Reports

Papers and Theses Produced November 1984 to June 1985

Papers

1. Jalote P. and R. H. Campbell, "Atomic Actions in Concurrent Systems," Proceedings of the *5th International Conference on Distributed Computing Systems*, Denver, May 1985.
2. Jalote P. and R. H. Campbell, "Fault Tolerance using Communicating Sequential Processes," *14th International Conference on Fault-Tolerant Computing (FTCS-14)*, Orlando, Florida, June 1984, pp. 347-352. Also accepted for publication in IEEE Transactions on Software Engineering, Special Issue, Software reliability, November 1985.
3. Liestman, A. and R. H. Campbell, "A Fault Tolerant Scheduling Problem," (*Accepted for publication in IEEE Transactions on Software Engineering.*) 1985.
4. Campbell R. H. and B. Randell, "Error Recovery in Asynchronous Systems," UIUCDCS-R-83-1148, Dept. Computer Sci., Univ. Illinois, 1983.
   Accepted for publication in IEEE Transactions on Software Engineering, December 1985.

Theses

5. Grunwald, D. C., "An Implementation of Path Pascal," MS Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1985.
6. Jalote P., "Atomic Actions in Concurrent Systems," Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1985.

Papers and Theses Produced Before October 1984

Papers

7. Balocca, R. and R. H. Campbell, "PP-11, A Path Pascal Language System for the PDP-11," *Proceedings of the Eighth Texas Conference on Operating Systems*, Dallas, November, 1979.
8. Campbell, R. H. and R. B. Kolstad, "Path Expressions in Pascal," *Proceedings of the Fourth International Conference on Software Engineering*, Munich, September 17-19, 1979, 212-219.
9. Campbell, R. H. and R. B. Kolstad, "Practical Applications of Path Expressions to Systems Programming," *ACM79*, Detroit, 1979, 81-87.
10. Campbell, R. H., K. Horton, and G. G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," *Digest of Papers FTCS-9: Ninth Annual International Symposium on Fault-Tolerant Computing*, Madison WI, June, 1979, 95-102.
11. Horton, K. H., R. H. Campbell, and G. G. Belford, "Meeting Real-time Deadlines," *Proceedings of Computers, Electronics and Control*, 1978, ACTA Press, Calgary,

1979.

12. Campbell, R. H. and R. B. Kolstad, "An Overview of Path Pascal's Design," *Sigplan Notices*, Vol. 15, No. 9, pp. 13-14, September, 1980.
13. Kolstad, R. B. and R. H. Campbell, "Path Pascal User Manual," *Sigplan Notices*, Vol. 15, No. 9, pp. 15-24, September, 1980.
14. Kolstad, R. B. and R. H. Campbell, "Direction for User Defined Communication for Distributed Software," *Proceedings of The International Conference on Parallel Processing*, IEEE 80CH1569-3, pp. 188-189, Boyne, MI, August 26-29, 1980.
15. Wei, A. Y., K. Hiraishi, R. Cheng, R. H. Campbell, "Application of the Fault-Tolerant Deadline Mechanism to a Satellite On-Board Computer System," *Digest of Papers FTCS-10: Tenth International Symposium on Fault-Tolerant Computing*, Kyoto, Japan, October 1980.
16. Cheng, W. Y., S. Ray, R. Kolstad, J. Luhukay, R. Campbell, and J. W-S. Lui, "ILLINET-A 32 Mbits/sec. local-area network," *Proceedings of the 1981 National Computer Conference*, Chicago IL, May 1981, pp209-214.
17. Campbell, R. H., "Distributed Path Pascal," *In Distributed Computing Systems*, (Editor Y. Paker and J.-P. Verjus), Academic Press, 1983, pp191-224.
18. Liestman, A. and R. H. Campbell, "A Fault Tolerant Scheduling Problem," *Digest of Papers FTCS-13: Thirteenth Annual International Symposium on Fault-Tolerant Computing*, Milano Italy, June 1983.
19. Jalote P., "Specification and Testing of Abstract Data Types,".*Proc. of CONCOM*, 1983.
20. Campbell, R. H. and T. Anderson, "Practical Fault Tolerant Software for Asynchronous Systems," *SAFECOMP 83, Third International IFAC Workshop on Achieving Safe Real-time Computer Systems*, Pergamon Press, Oxford, England, 1983.
21. Mickunas, M. D., P. Jalote and R. H. Campbell "The Delay/Re-read Protocol for Concurrency Control in Databases," *Proceedings of the First International Conference on Computer Data Engineering (COMPDEC)*, Los Angeles, April 1984, pp. 307-314.
22. McKendry, M. S., and R. H. Campbell, "A Mechanism for Implementing Language Support in High-Level Languages," *Transactions on Software Engineering*, Vol. SE-10, No. 3, May 1984, pp.227-236.

Theses

23. Balocca, Richard Joseph. "PP-11, A Path Pascal Compiler System," M.S. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1980.
24. Donnelly, Jeffrey M. "Porting the Newcastle Connection to 4.2 Berkeley UNIX," M.S. Thesis, Department of Computer Science Technical Report #1199, Universi-

ty of Illinois at Urbana-Champaign, Urbana, Illinois, 1985, p. 37.

25. Grass, Judith Ellen. "On Tcode Generation in a Pascal Compiler," M.S. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1982, p. 171.
26. Horton, Kurt H. "A Fault-Tolerant Deadline Mechanism," M.S. Thesis, Department of Computer Science Technical Report #998, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1979, p. 52.
27. Kolstad, R. B.. "Distributed Path Pascal: A Language for Programming Coupled Systems," Ph.D. Thesis, Department of Computer Science Technical Report #1136, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1983, p. 70.
28. Liestman, Arthur L. "Fault-Tolerant Scheduling and Broadcast Problems," Ph.D. Thesis, Department of Computer Science Technical Report #1063, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1981, p. 98.
29. McKendry, M. S.. "Pathos: An Experiment to Evaluate Path Pascal," M.S. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1980, pp. -.
30. McKendry, M. S.. "Language Mechanisms for Context Switching and Protection in Level Structured Operating Systems," Ph.D Thesis, Department of Computer Science Technical Report #1078, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1981, p. 143.
31. Schmidt, George Joseph. "The Recoverable Object as a Means of Software Fault Tolerance," MS Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1983.
32. Wei, Anthony Yu-Wu. "Real-Time Programming with Fault-Tolerance," Ph.D. Thesis, Department of Computer Science Technical Report #1041, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1981, p. 125.

Technical Reports

33. Campbell, R. H. and T. J. Miller, "A Path Pascal Language," UIUCDCS-R-78-919, Dept. Computer Sci., Univ. Illinois, Champaign-Urbana, 1978.
34. Campbell, R. H. and R. B. Kolstad, "A Practical Implementation of Path Pascal," UIUCDCS-R-80-1008, Dept. Computer Sci., Univ. of Illinois at Urbana, 1980.
35. Cheng, W. Y., S. Ray, R. Kolstad, J. Luhukay, R. Campbell, J.W.-S. Liu, "ILLINET-- A 32 Mbits/sec. Local Area Network," UIUCDCS-R-80-1035, Tech. Report, Dept. Computer Science, Univ. Illinois, 1980.
36. Hiraishi, K., A. Y. Wei, R. Cheng, R. H. Campbell, "Simulation of a Satellite On-Board Computer System Using Extended Path Pascal," UIUCDCS-R-80-1041, Dept. Computer Sci., Univ. Illinois at Urbana, 1980.
37. Kolstad, R. B. and R. H. Campbell, "Path Pascal User Manual," UIUCDCS-R-80-893, Dept. Computer Sci., Univ. Illinois at Urbana, February, 1980.

38. Kolstad, R. B. & R. H. Campbell, "User Defined Communication For Distributed Software," UIUCDCS-R-80-1015, Dept. Computer Sci., Univ. Illinois at Urbana, March, 1980.

39. Liestman, A. and R. H. Campbell, "A Fault Tolerant Scheduling Problem," UIUCDCS-R-80-1010, Dept. Computer Sci., Univ. Illinois at Urbana, 1980.

40. McKendry, M. S. and R. H. Campbell. "The Execute Statement: Design, Examples, and Implementation Algorithms," UIUCDCS-R-80-1044, Dept. Computer Sci., Univ. Illinois, 1980, p. 22.

41. McKendry, M. S. and R. H. Campbell. "Mechanisms for Protection and Process Control in Operating System Languages," Department of Computer Science Technical Report #1038, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1980, p. 15.

42. McKendry, M. S., R. H. Campbell and R. B. Kolstad, "PATHOS: A Path Pascal Operating System," UIUCDCS-R-80-1016, Dept. Computer Sci., Univ. Illinois at Urbana, 1980.

43. McKendry, M. S. and R. H. Campbell, "Mechanisms for Protection and Process Control in Operating System Languages," UIUCDCS-R-80-1038, Dept. Computer Sci., 1980.

44. McKendry, M. S. and R. H. Campbell, "Capabilities for High Level Languages," UIUCDCS-R-80-1039, Univ. Illinois, Dept. Computer Sci., December, 1980.

45. Wei, A. Y. and R. H. Campbell "Construction of a Fault-Tolerant Real-Time Software System," UIUCDCS-R-80-1042, Dept. Computer Sci., Univ. Illinois at Urbana, 1980, p. 24.

46. Wei, Anthony Y., K. Hiraishi, R. Cheng, R. H. Campbell, "Application of the Fault-Tolerant Deadline Mechanism to a Satellite On-Board Computer System," UIUCDCS-R-80-1012, Dept. Computer Sci., Univ. Illinois at Urbana, 1980.

47. Mcnabb, D. A., G. M. Beshers, R. H. Campbell, "TCODE: An Intermediate Code for the Path Pascal Compiler," UIUCDCS-R-82-1060, Dept. Computer Sci., Univ. Illinois, 1982.

48. Jalote P. and R. H. Campbell, "Fault Tolerance using Communicating Sequential Processes" UIUCDCS-R-83-1149, Dept. Computer Sci., Univ. Illinois, 1983.

49. Mickunas, M. D. and Jalote P., "The Delay/Re-Read Protocol for Concurrency Control in Databases," Department of Computer Science Technical Report #1145, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1983, p. 13.

50. Mickunas M. D., P. Jalote and R. H. Campbell, "A New Protocol for Concurrency Control Using Both Preventive and Corrective Techniques," *Technical Report*, UIUCDCS-R-84-1163, 1984, p. 23, (Submitted to *IEEE Transactions on Software Engineering.*)

APPENDIX B

Atomic Actions in Concurrent Systems

P. Jalote and R. H. Campbell,

Proceedings of the
5th International Conference on Distributed Computing Systems,
Denver, May 1985.

Atomic Actions For Fault Tolerance Using CSP[1]

Pankaj Jalote
Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract:

Two complementary techniques have evolved for providing fault tolerance in software: *forward error recovery* and *backward error recovery*. Few implementations permit both approaches to be combined within a particular application. Fewer techniques are available for the construction of fault-tolerant software for systems involving concurrent processes and multiple processors. Many schemes for supporting forward or backward recovery are based on some concept of an atomic action. In this paper, we propose a mechanism for supporting an atomic action in a system of Communicating Sequential Processes (CSP). The atomic action is used as the basic unit for providing fault tolerance. The atomic action is called an FT-Action, and both forward and backward error recovery are performed in the context of an FT-Action. An implementation for the FT-Action is proposed, which employs a distributed control, uses CSP primitives, and supports local compile and run-time checking of the forward and backward error recovery schemes.

[1]This work was supported in part by a grant from NASA, NSG1471.

1. Introduction

Several practical techniques for the construction of fault-tolerant software have evolved [23]. The aim of these techniques is to ensure that the system provides the intended service despite possible software or hardware faults. The techniques depend upon two complementary approaches to fault-tolerance known as *forward error recovery* and *backward error recovery* and it has been suggested that both be used to provide more reliable software [2, 7, 8].

Forward error recovery aims to identify the error and, based on this knowledge, correct the system state containing the error [4]. The approach requires accurate damage assessment and identification of the cause of the error. *Exceptions, signal and raise operations, and exception handlers* are common mechanisms used to provide forward recovery [2, 17]. In contrast, *backward error recovery* corrects the system state by restoring the system to a state which occurred prior to the manifestation of the fault. The *recovery block scheme* [22] provides a system structure that supports backward recovery. The scheme involves *acceptance tests, recovery points* and *alternate* algorithms.

Proposed extensions of the recovery block scheme to an environment involving communicating concurrent processes must solve the problem that any exchange of information may propagate an error from one process to another. If communications are not coordinated with recovery points, backward error recovery may create an uncontrolled rollback of many processes called the *domino effect* [22]. A language construct called a *conversation* [22] has been proposed to provide a static backward error recovery scheme for concurrent processes which prevents the domino effect. This is in contrast to the approach of determining the amount of rollback needed dynamically [15, 21, 28]. Each process that joins a conversation has a recovery point, an acceptance test, and alternate algorithms. While a process is in a conversation, it may only communicate with other processes in the same conversation. If *any* process fails an acceptance test or oth-

erwise detects an exception, every process in the conversation performs a rollback to its recovery point, established on entry to the conversation, and uses an alternate algorithm. This restriction on communication limits the propagation of errors and eliminates the possibility of the domino effect. Several implementations of conversations have been described [6, 25, 26].

Forward error recovery in systems of communicating processes is discussed by Campbell and Randell in [7]. A framework for exception handling is proposed that is based on the use of *atomic actions*. Using their definition of an atomic action, a process participating in an atomic action may only exchange information with other processes in that atomic action. If any process raises an exception, every process in the atomic action invokes an exception handler for the exception. If several exceptions are raised concurrently, an exception resolution scheme selects a single exception to represent the combination of the exception conditions.

Exceptions are organized into a tree or, for generality, a lattice structure in which the upper bound is the *universal exception*. The resolution algorithm selects the exception which is the least upper bound in the tree of all of the exceptions that have been raised.

If all the processes can recover, the processes return from the exception handlers and complete the atomic action normally. However, if any of the processes cannot recover, all of the processes complete the atomic action abnormally and signal[1] an exception. If the processes do not agree upon the exceptions they signal, a failure exception is signaled. Backward error recovery, recovery blocks, and conversations can be implemented as particular applications of the forward error recovery scheme.

Many existing programming languages such as Ada, CLU, PL/1 and Mesa include forward error recovery facilities based on some form of an exception handling approach [2]. However, few

[1] A raised exception within an atomic action is used to invoke local exception handlers. An atomic action returns a signaled exception in order to raise that exception at the point of invocation of the action.

of these mechanisms are proposed as general solutions to the programming of forward error recovery in concurrent processing systems. Several programming languages such as Argus [18] and many language extensions [1, 16, 17, 20, 26] have been proposed to permit backward error recovery in concurrent processing systems. The provision of both forward and backward error recovery facilities within the same concurrent programming language remains a problem. New techniques to allow the use of complementary forward and backward error recovery schemes within the same concurrent programming language are needed [6].

In this paper, we propose a scheme to support backward and forward error recovery in a system of Communicating Sequential Processes (CSP) [12], based on the framework of atomic actions. We have chosen to use CSP because its processes do not share memory, all communication between its processes must be programmed explicitly, many problems caused by side-effects in concurrent programming languages are avoided, program verification is simplified, and it is well-known. We present a notation for specifying a planned atomic action, called an FT-Action. The FT-Action includes support for both backward and forward error recovery so that they may be used in a complementary way. The basic FT-Action scheme can be implemented using only CSP primitives. The recovery control is distributed over the processes taking part in the communication.

This paper is organized as follows. First, we briefly describe CSP and the concept of atomic actions and then we introduce the FT-Action and discuss how it may be used to support error recovery. Next, we explain how the FT-Action can be implemented in CSP. We discuss the advantages and disadvantages of the proposed scheme and its implementation. Finally, we review our decision to use CSP.

2. Communicating Sequential Processes

CSP was proposed by Hoare as the basis for a concurrent programming language. CSP uses Dijkstra's guarded commands [9] as sequential control structures, and as the sole means of introducing and controlling nondeterminism. A parallel command specifies concurrent execution of its constituent sequential commands (processes). All the processes start simultaneously, and the parallel command terminates successfully only if and when they all have successfully terminated. Simple forms of input and output commands are introduced which are used for communication between the concurrent processes. CSP processes may only communicate with each other using the input and output message commands. Messages are passed to named processes through synchronous *static channels*. An *output command* is of the form:

destination ! expression

where *destination* is the process name and *expression* is a simple or structured value. An *input command* has the form:

source ? target

where *source* is a process name and *target* is a simple or structured variable.

Communication occurs between two processes of a parallel command whenever (1) an input command in one process specifies as its source the process name of the other process; (2) an output command in the other process specifies as its source the process name of the first process; and (3) the target variable of the input command matches the value denoted by the expression of the output command. On these conditions, the input and output commands are said to *correspond*. An input command fails if its source is terminated. An output command fails if its destination is terminated or if its expression is undefined.

Commands which correspond are executed simultaneously, and their combined effect is to assign the value of the expression of the output command to the target variable of the input command. There is *no* automatic buffering, and an input or output command is delayed until the other process is ready with the corresponding output or input command. A communicating process may wait forever if another process does not match its command. This inherent limitation of a synchronous message passing scheme makes detection of a so called "deserter" [16] or dead process difficult. After the communication, both processes proceed independently and concurrently.

Dijkstra's Guarded Commands [9] are used in CSP in the form:

$$G \rightarrow C$$

where G is a guard consisting of a list of boolean expressions followed by an optional input command list, and C is a command list. Output commands may not appear in the guards. If an input command appears in a guard, it is called an input guard. A guarded command may be executed if and when the execution of its guard does not fail. First, the guard is evaluated by determining the value of its boolean expressions. If any expression is false, the guard fails; but a guard that evaluates to true has no effect. An input guard may be evaluated only if and when there is a corresponding appropriate output.

The alternative command may be executed by a sequential process. It has the form:

$$[\ G_1 \rightarrow C_1 \ \square\ G_2 \rightarrow C_2 \ \square\ \ldots \ \square\ G_n \rightarrow C_n\ ]$$

and selects the execution of exactly one of the constituent guarded commands. If all the guards fail, the alternative command fails. Otherwise a command is selected non-deterministically from those commands with successful guards. In the case when more than one command list can be

selected, the choice is nondeterministic. If several of the input guards in an alternate command correspond with output commands elsewhere, only one is selected and the others have no effect.

The notation $(i{:}1..n)G \rightarrow C$ represents the alternative command

$$[\ G_1 \rightarrow C_1 \ \square\ G_2 \rightarrow C_2 \ \square\ \ldots \ \square\ G_n \rightarrow C_n\ ]$$

where each $G_j \rightarrow C_j$ is formed from $G \rightarrow C$ by replacing every occurrence of the bound variable i by the numeral j.

A repetitive command specifies as many iterations as possible of its constituent alternative command. It has the form:

* [alternative command]

When all the guards fail, the repetitive command terminates. Otherwise, the alternative command is executed once and the whole repetitive command is executed again. A repetitive command may have input guards. If all the sources named by the input guards have terminated, then the repetitive command also terminates.

It is possible to program coroutines using the CSP notation, and consequently subroutines can also be programmed. The provision of output commands within the guards has been advocated in [3, 27]. We will assume a version of CSP with both this facility and a basic exception mechanism for a single process.

3. Atomic Actions and Fault Tolerance

Most of the techniques for structuring systems deal with the organization and sub-division of the static (or spatial) structure of the system. However, the pattern of interaction between the components of a system is also of interest, particularly for providing fault tolerance. Interactions

between components reflect the dynamic (or temporal) structure of the system. The concept of atomic actions can be used to structure the temporal activity of the system.

An atomic action is an activity, possibly consisting of many steps performed by many different processors, that appears primitive and indivisible to any activity outside the atomic action. To other activities, an atomic action is like a primitive operation which transforms the state of the system without having any intermediate states. Operations which are strictly sequenced, non-interfering, and non-overlapping with respect to other operations are often said to be executed as atomic actions.

Atomic actions may be *planned atomic actions.* Such atomic actions are identified during the design of the system and are supported by some run-time mechanism. Atomic actions may also be *dynamically identified atomic actions* [5]. These may only be discovered by examining the system's history of execution. There are two different views of planned atomic actions [14] and we refer to these as *recoverable atomic actions* [1, 18, 24], and *basic atomic actions* [2, 20].

Recoverable atomic actions conform to the "all or nothing" view, which requires that either all the objects modified by the action change to their final state, or all of the objects remain in their initial state. Recoverable atomic actions specify that both *indivisibility* and *recoverability* [18] are fundamental requirements for atomicity.

Haerder and Reuter's paper on database transactions [11] distinguishes the properties of transaction consistency and atomicity. Together, consistency and atomicity imply recoverable atomic actions. The imposition of recoverable atomic actions as an obligatory fault tolerance mechanism may be useful in some contexts, but may result in inflexibility when different recovery techniques are desired. For example, forward error recovery using exception handling may be required.

In basic atomic actions, indivisibility is the only requirement of atomicity, and recoverability is not considered a necessary part. Where recovery is desired, it is constructed using basic atomic actions. This approach allows the flexibility to use different recovery techniques. For the rest of this paper we will use the term atomic actions to mean basic atomic actions.

Many techniques for supporting fault tolerance have used the property of atomicity. The conversation block [22] has the restriction that there may be no interaction across the boundaries of the conversation. This restriction guarantees the atomicity of the computation performed inside the conversation. Similarly, the recovery block [22] in sequential systems forms an atomic action with recovery primitives. Dynamic techniques for backward recovery [15, 21, 28] aim to identify that part of the computation which had no interaction with the rest of the computation, thereby employing dynamically identified atomic actions. The use of atomic actions for forward error recovery is proposed by Campbell and Randell [7].

The different proposals in the literature have made it clear that atomic actions are useful for recovery. We believe that atomic actions are not merely desirable, but are necessary for recovery [13]. Hence, we have extended CSP with an atomic action called an FT-Action within which forward and backward error recovery can be programmed for concurrent processes.

4. Design of an FT-Action

An FT-Action should be designed so that its atomicity is guaranteed. The atomicity guarantee permits the programming of recovery for the construct. The scheme should be able to support both the programming of backward error recovery and forward error recovery. For backward recovery we employ a conversation-like scheme, which can be easily implemented in a planned atomic action framework. For forward recovery, we use the scheme proposed by Campbell and Randell [7] which is based on planned atomic actions.

We define a *Fault Tolerant Atomic Action (FT-Action)* as a distributed control structure that a group of processes may join or leave together in synchrony. Inside an FT-Action the processes may communicate with one another, but not with processes outside of the control structure. The FT-Action will be used as a framework within which error recovery can be provided. It has the following properties:

Atomicity: To guarantee atomicity and prevent *information smuggling* [16], the communications of processes in the control structure must be isolated from other processes. Hence, in an FT-Action, no communication may take place across the boundaries of the FT-Action. The corresponding command for an input or output command inside an FT-Action must also be inside that FT-Action.

A recovery line for backward error recovery: In the event of an error, the processes may be rolled back to the recovery points that were established at the recovery line. An FT-Action provides a recovery line which is defined by the synchronized entry of all the participating processes.

A test line for the processes: The test line is a set of diagnostic tests, one for each process, which is used to determine whether any errors have occurred. In an FT-Action the exit statements (see next section) in the constituent processes together form a test line. The processes synchronize at the test line.

Recovery measures: An FT-Action should include facilities and primitives for recovery. If any process inside an FT-Action detects an error which cannot be corrected locally, it is an error for the entire FT-Action and *all* the processes taking part in the FT-Action must cooperatively invoke appropriate recovery measures.

Nesting of FT-Actions: FT-Actions may be nested, but only strict nesting is permitted.

Strict nesting allows hierarchical recovery techniques so that if recovery is unsuccessful in an FT-Action, it may be attempted in an enclosing FT-Action.

As a practical point, an implementation ought to detect and allow recovery from a *deserter process* [16]. In CSP, this can occur if an input or output command is never matched or a process dies. This may also occur if a process which is expected to participate in an FT-Action does not. Whereas the detection of such an exception is simplified by a shared data system, it is especially difficult in a message passing system since a process cannot unilaterally observe the state of another process.

5. Error Recovery with the FT-Action

In this section, we give the notation for a basic FT-Action and describe the primitives needed. We outline how backward, forward and combined recovery may be implemented using FT-Action primitives. The primitives have CSP implementations which are described in the following section.

5.1. FT-Action Primitives

A FT-Action is identified by a declaration which includes a list of all the processes which will participate in the FT-Action. Each process taking part in an FT-Action must declare a corresponding FT-Action entry statement. The name and lists supplied by each of the processes taking part in the FT-Action are compared at run-time to ensure consistency. A possible syntax

for an FT-Action is shown in Figure 5.1.

```
A : FT-Action with (P1,P2,...,Pn)
...
P1 ::[  ...
        FT-Action A
                <code>
                exit unless <e>
                <code>
                exit unless <e>
                ...
        end
 ]
```

Figure 5.1: The FT-Action.

The FT-Action synchronizes recovery schemes involving the processes $P_1, P_2, \cdots, P_n$. Each process should contain a statement similar to that declared in P_1. The body of the FT-Action includes "exit" statements, each of which corresponds to a test point within a test line. When a process reaches an exit, it waits for the other processes in the FT-Action to reach their corresponding exits. The exception "e" of a test line is evaluated by an interprocess voting scheme (described in the next section). This scheme combines exceptions detected by the processes in the FT-Action using exception resolution [7] and distributes the resulting exception value to each process. Thus, the test line returns the same exception to each process. In the case that no process detects an exception, the scheme produces a null value. If the exception is null, then the FT-Action is successful and the exit statement terminates the structure. Otherwise, the processes continue in the FT-Action and invoke recovery measures. An FT-Action can terminate abnormally by signaling an exception.

To ensure the atomicity of the FT-Actions, for the duration of the FT-Action, the process P_1 only communicates with the processes mentioned in the declaration of the FT-Action. That is, within the body of the FT-Action, an input or output command in P_1 may only have a process $P_2, P_3, \ldots, P_n$ as the source or destination process, respectively.

In general, implementation of either the forward or backward error recovery scheme will require the use of several exit primitives. In the rest of the figures in this section, we will omit the body of the process P_1 enclosing the FT-Action.

5.2. Backward Recovery

Informally, an FT-Action may be used to specify backward error recovery as shown in Figure 5.2.

```
FT-Action A
        ensure <acceptance test>
        by     <primary>
        else by <alternate>
               ...
        else by <alternate>
        else signal error
end
```

Figure 5.2: Backward error recovery structure.

Backward error recovery constitutes a Conversation [22] involving the processes $P_1, P_2, ..., P_n$. Each process executes its primary and may communicate with other processes executing their primaries in the FT-Action. Each process then evaluates its acceptance test. An exception is raised if an acceptance test fails or a run-time error is detected. If no exception is raised, the FT-Action terminates. If an exception is raised by any of the processes, then every process invokes backward error recovery. Every process in an FT-Action is required to have the same number of alternates. The FT-Action synchronizes execution of the alternates so that each process keeps in step. The processes may communicate with one another during the execution of an alternate.

The backward error recovery structure shown in Figure 5.2 can be transformed into the FT-Action primitives shown in Figure 5.3.

```
FT-Action A
        <save state>
        <primary; acceptance test>
        exit unless <e>
        <restore state>
        <alternate; acceptance test>
        ...
        exit unless <e>
        <restore state>
        signal error
end
```

Figure 5.3: FT-Action backward error recovery.

The state (values of the variables) of each process is saved[1] after it enters the FT-Action. The saved states of all the processes correspond to the recovery points that form the recovery line. Before the first test line (first set of exit statements), each process evaluates its acceptance test to detect exceptions. If one or more processes detect exceptions, the exception "e" returned by the test line will not be null and the exit statements will not terminate the construct. Instead, the processes roll back and execute their next alternates. After reevaluating their acceptance tests, the processes reach another test line. This sequence is repeated until either the exception returned by a test line is null or the last alternates are attempted. The last alternates are used to "signal" an exception to indicate that the FT-Action has failed.

5.3. Forward Recovery

To specify forward error recovery using an FT-Action, the notation presented in [8, 19] is used with some modification. Informally, the FT-Action may be used to specify forward error recovery as shown in Figure 5.4.

[1] A "discard state" operation [7] is unnecessary in CSP, but might be required in other languages. It could be added to the scheme by extending the FT-Action with a finalization clause that is executed after the exit is taken.

```
FT-Action A
        <algorithm>
                [ excep e → handler (e) ]
end
```

Figure 5.4: Forward error recovery structure.

This FT-Action coordinates forward error recovery for the processes $P_1, P_2, \ldots, P_n$. It terminates if no exceptions are raised during the execution of the algorithms. If an exception is raised by any process, then *all* the processes in C are notified of the exception. Every process will then execute its exception handler ("handler (e)") for that exception. Forward error recovery completes when every process either "signals" an exception or successfully completes its handler. If an exception is signaled in an FT-Action, the action terminates abnormally with an exception. If the FT-Action is nested within another, the exception is raised in the containing FT-Action.

In general, fault-tolerance provisions should be localized as much as possible to prevent costly error propagation and repair [2]. The ability to be able to nest FT-Actions permits the programmer to devise forward error recovery schemes that minimize the number of processes that need to be involved in responding to the detection of an exception. Forward error recovery for a single process can be considered as a special case of an FT-Action.

The forward error recovery can be translated into the primitives shown in Figure 5.5.

```
FT-Action A
        <primary>
        exit unless e
        [
                e = my_signal          → signal e
                □ e ≠ my_signal        → handler(e)
        ]
        exit unless e
        [
                e = my_signal          → signal e
                □ e ≠ my_signal        → signal error
        ]
end
```

Figure 5.5: FT-Action error forward recovery.

The first test line after the primaries resolves any raised or signaled exceptions[1] during the execution of the primaries To prevent ambiguity, we require that no exception may be both raised and signaled in the same FT-Action. If the resolution scheme is applied to a raised and a signaled exception, the exception tree should guarantee that the scheme will raise a failure exception.

If the exception "e" returned by the test line is null, the exit statement terminates the FT-Action. If each process locally signaled "e", the FT-Action terminates by signaling "e". Otherwise, each process attempts recovery by executing its handler for "e".

The handlers may, in turn, raise or signal exceptions. When the processes complete their handlers, the exception "e" for the second test line is determined. If "e" is a null exception, the exit statement terminates the FT-Action. If each process locally signaled "e" within its handler, the FT-Action terminates by signaling "e". Otherwise, the FT-Action signals error.

[1] A raised exception indicates that an exception was detected within a primary and requires handling within the FT-Action, whereas a signaled exception indicates that a primary intends to return an exception to an enclosing FT-Action [19].

5.4. Combined Recovery

Forward and backward error recovery schemes may be combined. One method of using the two techniques in a complimentary manner is shown in Figure 5.6.

```
FT-Action A
        ensure <acceptance test>
        by <primary>
                [ excep e → handler (e) ]
        else by <alternate>
                ...
        else by <alternate>
        else signal error
end
```

Figure 5.6: Forward and backward error recovery structure.

In Figure 5.6, a forward recovery scheme is associated with the primary algorithm and would be invoked for the specified exceptions. The backward error recovery scheme would be invoked for other exceptions and any exceptions that might occur in the handler. There are many ways to combine forward and backward error recovery schemes. Such combinations may be transformed into pri iitives as before.

The forward error recovery structure shown in Figure 5.6 can be transformed into the FT-Action primitives shown in Figure 5.7.

```
FT-Action A
        <save state>
        <primary; acceptance test>
        exit unless e
        [
                e = my_signal  → signal e
                □ e ≠ e → handler(e)
                □ e ≠ my_signal; e ≠ e  → skip
        ]
        <acceptance test>
        exit unless e
        <restore state>
        <alternate; acceptance test>
        ...
        exit unless e
        <restore state>
        signal error
end
```

Figure 5.7: FT-Action forward and backward error recovery.

Each process completes its primary and evaluates its acceptance test. If the exception "e" returned by the first test line is the null exception, the exit statement terminates the FT-Action. If each process locally signaled "e", the FT-Action terminates by signaling "e". In this case, we assume that the signal is a valid result of the primary and passes the acceptance test. Otherwise, each process attempts recovery by executing the handler for "e".

The exception for the second test line is determined after the reevaluation of the acceptance tests. This time, if the exception is not null, backward error recovery is applied and the processes execute their next alternates. Although the FT-Action does not impose any implementation restriction, we choose to simplify recovery strategies by transforming any exceptions which are signaled from a handler into an error.

5.5. An Example

To illustrate the use of FT-Actions, we present a simple example. There are two processes, P and Q. P computes n different values, and sends the computed data to Q. Q constructs

records using the data sent by P, sorts the records, and then stores them in a file. Two nested FT-Actions provide fault tolerance. FT-Action A encompasses the whole activity and provides fault tolerance based on backward error recovery. The FT-Action B is nested within A, and supports forward error recovery for the construction of each record.

The backward error recovery scheme for Q employs two methods to produce a sorted file of records. The primary method inputs data, constructs a record, and inserts it in its proper place in a sorted file. The alternate method inputs data, constructs a record, and appends it to an unsorted file. After all the data has been received, the file is sorted. The backward recovery scheme for P uses the same algorithm for both the primary method and the alternate method.

The process Q may raise an exception in constructing the record from the data that it received. We refer to this exception as a transmission exception. The FT-Action B provides forward error recovery for this exception. This example is shown in Figure 5.8.

```
A : FT-Action with (P, Q)
B : FT-Action with (P, Q)

P :: [ ...
     FT-Action A
        ensure (true) by
        [    i := 0;
             *[ i < n →
                  i := i+1;
                  FT-Action B
                       [    Compute (data);
                            Q ! data;
                       ]
                            [except transmission →
                                 Q ! data;
                            ]
                  end
             ]
        ]
        else by
        [    i := 0;
             *[ i < n →
                  i := i+1;
                  Compute (data)
                  Q ! data;
             ]
        ]
        else signal error
     end
     ...
]

Q :: [ ...
     FT-Action A
        ensure (file sorted) by
        [    i := 0;
             *[ i < n →
                  i := i+1;
                  FT-Action B
                       [    P ? data;
                            Compute rec using data;
                       ]
                            [except transmission →
                                 P ? data;
                                 Compute rec using data;
                            ]
                  end
                  Insert rec in the file;
             ]
        ]
        else by
        [
             i := 0;
             *[ i < n →
                  i := i+1;
                  P ? data;
                  Compute rec using data;
                  Store rec in file;
             ]
             Sort the file;
        ]
        else signal error
     end
     ...
]
```

Figure 5.8: An example using nested FT-Actions.

The example illustrates several important aspects of forward error recovery. In FT-Action *B*, although Q detects the exception, P must also take part in the recovery. Consequently, the exception applies to the entire FT-Action *B*, and it is irrelevant which process within *B* detects the error. Accordingly, it is not possible to determine by examining the program which process detected the error. Similarly, even though process P passes its 'acceptance test' in FT-Action *A*, P must perform recovery if process Q fails the acceptance test in FT-Action *B*.

The primaries and alternates do not need a similar structure. In this example, the primaries of FT-Action A contain a nested FT-Action B, while there are no nested actions for the alternates of action A. If a transmission exception is raised in these alternates, the FT-Action A will fail.

Fault tolerance provisions may lead to a loss of concurrency. In this example, the alternates for FT-Action A allow P and Q to compute the data and the record concurrently. However, there is less concurrency possible in the primaries of A because of the action B nested within A. The exit of FT-action B imposes an additional synchronization constraint on the processes P and Q.

6. Implementation

This section describes an implementation of the FT-Action primitives. The implementation uses only CSP primitives for communication and synchronization between processes. The reliability of the recovery schemes is enhanced by compile and run-time checking.

A combination of compile and run-time checking is used to prevent information smuggling. The C-Set of a process for an FT-Action is the set containing the name of the conversation and the names of the processes specified in the FT-Action declaration. A syntactic check ensures that, while inside an FT-Action, a process only communicates to the other processes identified in the C-Set of that action. A further run-time check must be used to ensure that the C-Sets of the processes involved in a particular FT-Action are the same.

The correct nesting of FT-Actions can be checked at compile-time by examining each process. Each process identifier that occurs in the statement of a nested FT-Action must also occur in the statement of any enclosing FT-Action.

An FT-Action can be transformed into CSP primitives by a preprocessor. For the purposes of implementation, we require that the processes within an FT-Action have a static ordering (for example, we use the lexicographic ordering defined by their identifiers).

6.1. FT-Action Entry

The entry of a process into an FT-Action requires synchronization and a C-Set consistency check. The consistency check uses a voting technique based on the Two Phase Commit protocol [10]. Voting is implemented by passing a message up and down a chain of the processes attempting to enter the FT-Action.

The processes whose identifiers are included in the C-Set of an FT-Action are organized into a chain using their static ordering. Starting from the head of the chain, each process in a vote passes C-Set information to its successor. If the C-Set of any process does not agree with the information that the process receives, a C-Set exception is passed on. This ensures that the tail process will receive a C-Set exception if the C-Sets are not consistent. Next, the tail process returns the result of the vote back down the chain to the head. In this way, every process receives an exception if the C-Sets are inconsistent and the FT-Action is aborted.

The voting algorithm is shown in Figure 6.1. Different algorithms are used for the head, middle and the tail of the chain. Since the chain is constructed using the static ordering of the processes, a compile-time algorithm can construct the voting scheme. We assume that process P_i is the predecessor of process P_{i+1}.

For the head of the chain (process P_1):

P_2 ! C_Set;
[P_2 ? success () → skip
□ P_2 ? failure () → ABORT
]

For the middle of the chain (process P_i):

P_{i-1} ? C_Set ;
[(C_Set = My_C_Set) → P_{i+1} ! C_Set
□ (C_Set ≠ My_C_Set) → P_{i+1} ! C_Set_Exception
]
[P_{i+1} ? success () → P_{i-1} ! success ();
□ P_{i+1} ? failure () → P_{i-1} ! failure ();
ABORT
]

For the tail (process P_n):

P_{n-1} ? C_Set;
[(C_Set = My_C_Set) → P_{n-1} ! success () ;
□ (C_Set ≠ My_C_Set) → P_{n-1} ! failure () ;
ABORT
]

Figure 6.1: Transformation of the entry statement.

6.2. The Exit Statement

The exit primitive is used to terminate an FT-Action if it is successful. The implementation of the exit primitive also uses a chain-based voting scheme to decide whether an exception has been detected by any of the processes in the FT-Action. If an exception is detected, all the processes in the FT-Action must participate in recovery. Each process resolves any exception it may have received from a predecessor process with any exception it has raised and sends the result to its successor process. The final result is sent to each process in the FT-Action by transmitting it back down the chain. The "value" of an exception is null if no exception oc-

curred. The implementation scheme is shown in Figure 6.2.

For the head of the chain (process P_1):

P_2 ! my_exception;
P_2 ? final_exception →
    [final_exception = null → exit
    □ final_exception ≠ null → skip;
    ]

For the middle of the chain (process P_i):

P_{i-1} ? exception ;
P_{i+1} ! resolve(exception, my_exception) ;
P_{i+1} ? final_exception →
    P_{i-1} ? final_exception;
    [final_exception = null → exit
    □ final_exception ≠ null → skip
    ]

For the tail (process P_n):

P_{n-1} ? exception ;
final_exception := resolve(exception, my_exception);
P_{n-1} ! final_exception;
[final_exception = null → exit
□ exception ≠ null → skip
]

Figure 6.2 : Transformation of the exit statement

6.3. The Exception Mechanism

So far, the scheme that we have described only detects exceptions at the end of the primary or the alternates. However, a more desirable scheme would allow a process to raise an exception at a point other than at the end of a computation. Once an exception is raised, the processes in an FT-Action should not continue with the normal computation. Instead, all the processes should execute the exit statement and start the voting process. This could also happen if any process signals an exception to terminate the FT-Action.

Because of the synchronous message passing scheme of CSP, it is not possible simply to discontinue the normal computation of a process. Other processes which communicate in a normal manner with this process will wait indefinitely since corresponding input or output commands will not be executed.

A mechanism is required to notify all processes that an exception has occurred. On being notified of the exception, a process should start voting. The mechanism should also be capable of handling concurrent exceptions.

There are several ways to implement such a mechanism. For the purposes of this paper, we propose a simple scheme which only uses CSP primitives and requires a *broadcaster process* (BP) for each FT-Action. However, our scheme does require output commands in guards. A process that detects an exception communicates with the broadcaster process. The broadcaster process informs other processes taking part in the FT-Action that an exception has occurred.

The broadcaster process has two phases. In the first phase, it waits for input from any of the processes in the FT-Action. Any process which detects an exception, outputs an appropriate message to the broadcaster process. In the second phase, the broadcaster process tries to inform the other processes taking part in the FT-Action that an exception has been detected. The broadcaster process informs the processes only that an exception has occurred. The identity of the exception is still transmitted to the processes by the voting scheme. If more than one process detects an exception concurrently, then it will wait for the broadcaster process to input its message. Thus, during this second phase, the broadcaster process must also accept further exception messages. The broadcaster process is described in Figure 6.3.

```
[
    (i:1..n) P_i ? excep() → skip
]

*[
    (i:1..n) P_i ? excep() → skip
    □ (i:1..n) P_i ! excep() → skip
]
```

Figure 6.3: The broadcaster process (BP).

The processes taking part in the FT-Action must be able to input an exception from the broadcaster process as well as input and output to other processes. Thus, each input or output command of a process is transformed into an alternative command which may also input an exception message from the broadcaster process. If it receives an exception message, the process discontinues normal processing and starts the voting process. Otherwise it continues with the normal processing. A simplified transformation of an input or output command C in a process is shown in Figure 6.4. A detailed CSP implementation of this scheme would require the reorganization of statements and the addition of variables in order to reach an exit statement to start voting.

```
[
    BP ? excep() → start voting;
    □ C → skip;
]
```

Figure 6.4: Transformation of a command C.

A simple argument reveals the correctness of this scheme. If no exception occurs in the FT-Action, the command C will always be executed by all the processes as the broadcaster process will not be trying to output any exception message. Hence, the FT-Action will execute normally. If an exception occurs, a process which has not yet encountered the exception may reach the exit of the FT-Action, start voting, and detect an exception. Alternatively, it may try to

communicate with another process which has already encountered the exception and block. In this case, either the other process detected the exception and informed the broadcaster process or the other process received an exception message from the broadcaster process. As a consequence, the broadcaster process will either enter its second phase or will already be in its second phase. Thus, the process, if blocked, will receive an exception message from the broadcaster process which will allow it to proceed to vote. So, the processes always reach the voting phase, and the voting phase will then ensure that they are notified of the exception.

Note that the solution is complicated by non-deterministic communication. A process has a non-deterministic choice of executing C or receiving a message from the broadcaster process. Thus, it cannot be predicted exactly when or if an individual process will receive an exception message from the broadcaster process. However, it can be guaranteed that all the processes eventually reach the voting phase and the voting phase ensures that all the processes are informed of the exception.

The proposed scheme is simple and demonstrates that the exception mechanism can be implemented in the framework of CSP and FT-Actions. Other schemes can be designed with and without a broadcasting process. Similarly, other schemes can also be devised for implementing the entry and exit statements.

6.4. The Timeout Mechanism

The transformation scheme for the entry and exit statements has no mechanism to cope with the problem of a deserter process or certain design faults that could occur in an FT-Action. If a process is included in the C-Set of an FT-Action, but it does not enter the action (a deserter process), then it will block other processes from entering that action. This happens because its neighbors in the FT-Action voting chain will never be able to satisfy their I/O requests during

the first phase of the voting. There appears to be no satisfactory solution to this problem unless a timeout mechanism is provided in CSP. We describe a simple timeout mechanism which can be employed to solve the problem.

Each process starts a preset timer when it tries to communicate to its successor or predecessor process during the first phase of the voting process. This can be easily implemented using the CSP notation, by a simple transformation. If the I/O command C is an input or the output to process P_2, its simplified transformation is:

```
[
    C  →  skip
    □ timer = timeout  →  ABORT
]
```

otherwise its transformation is:

```
[
    C  →  skip
    □ timer = timeout  →  P_{i-1} ! failure (); ABORT
]
```

If a corresponding input/output for C is not executed within the set time, the process locally aborts the FT-Action. The timeout scheme assumes that output commands are allowed in the guards.

If there is a deserter process, then the first phase of voting cannot complete, and some process will timeout. A simple argument shows that if one process taking part in an FT-Action A times out, then all the processes for A will abort the FT-Action.

Suppose P_i is the first process to timeout. All the processes below P_i in the chain will receive a failure in the second phase of voting. The processes above P_i in the chain will timeout in

the first phase, since P_i will not send any vote to P_{i+1}. Hence, all the processes will eventually abort.

Using the simple scheme described here, a deserter process can be detected. However, the use of a timeout mechanism presents timing issues that might complicate the design of a reliable system. In any practical implementation of an FT-Action, a time-independent deserter detection mechanism would be more desirable.

7. Conclusion and Discussion

The paper proposes a notation to specify an atomic action for supporting fault tolerance in a system of Communicating Sequential Processes. CSP allows such a scheme to be described without concern for the implicit interprocess interactions that can occur in other, shared memory, concurrent programming languages. The atomic action is called an FT-Action and supports both forward and backward error recovery in an uniform manner. The control structure of an FT-Action is distributed over the processes taking part in it and is implemented using CSP primitives. The number of communication messages needed to coordinate the FT-Action is O(n), where n is the number of processes taking part in the FT-Action. The minimum number of communications needed is also O(n) since all processes must receive at least one message.

Although we have considered practical support for error recovery in concurrent systems, much further research and development is still required. Existing programming language support for error recovery either facilitates backward error recovery and concurrent processes as in Argus [18] or forward error recovery for a single process as in CLU [19]. Our proposal allows both forward and backward error recovery to be used as complementary mechanisms in systems of concurrent processes.

However CSP does impose difficulties in devising a comprehensive implementation of our scheme because of its symmetric communications, lack of timeouts, and restrictions on output commands in guards. Because of the message scheme of CSP, we have not been able to devise a simple strategy to detect a deserter process. System designers could make mistakes in designing communications that are difficult to detect at either compile-time or run-time. For example, input/output commands in an FT-Action must only match output/input commands in that FT-Action; failure to comply with this structure cannot be detected at compile time and can only be detected at run-time by a communication protocol time-out.

We believe that a structure like an FT-Action should be used in concurrent languages to provide both backward and forward error recovery support and to encourage the development of reliable concurrent applications. We have demonstrated the practicality of such an approach by devising a structure for CSP which can be transformed into CSP language primitives. CSP was not designed to support fault-tolerant software development. However, it does allow an exposition of many of the complex issues involved in constructing fault tolerance provisions for concurrent systems.

Acknowledgments

The authors wish to thank members of the Software Research Group at the University of Illinois and the anonymous referees for their constructive comments which greatly improved this paper.

References

[1] J. E. Allchin and M. S. McKendry, "Synchronization and Recovery of Actions," In: Proceedings of Symposium on Principles of Distributed Computing, ACM SIGACT-

SIGOPS, pp. 17-19, Montreal, 1983.

[2] T. Anderson and P. A. Lee, "Fault Tolerance, Principles and Practice." Prentice-Hall International, Englewood Cliffs, NJ, 1981.

[3] A. J. Bernstein, "Output Guards and Nondeterminism in Communicating Sequential Processes." *ACM TOPLAS,* vol. 2, no. 2, pp. 234-238, April 1980.

[4] E. Best and F. Cristian, "Systematic Detection of Exception Occurrences," *Science of Computer Programming,* vol. 1, no. 1, pp. 115-144, 1981.

[5] E. Best and B. Randell, "A Formal Model of Atomicity in Asynchronous Systems," *Acta Informatica,* vol. 16, pp. 93-124, 1981.

[6] R. H. Campbell, T. Anderson and B. Randell, "Practical Fault Tolerant Software for Asynchronous Systems," In: SAFECOM 83, Cambridge, 1983.

[7] R. H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems," UIUCDCS-R-83-1148, Department of Computer Science, University of Illinois at Urbana-Champaign, 1983.

[8] F. Christian, "Exception Handling and Software Fault Tolerance," *IEEE Transactions on Computers,* vol. C-31, no. 6, pp. 531-540, June 1982.

[9] E. W. Dijkstra, "Guarded Commands, Nondeterminancy and Formal Derivation of Programs," *Communications of the ACM,* vol. 18, no. 8, pp. 453-457, August 1975.

[10] J. N. Gray, "Notes on Database Operating Systems," In: Operating Systems: An Advanced Course, Lecture Notes in Computer Science, vol. 60, pp. 393-481, Springer-Verlag, New York, 1978.

[11] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *Computer Surveys,* vol. 15, no. 4, pp. 287-317, December 1983.

[12] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM,* vol. 21, no. 8, pp. 666-677, August 1978.

[13] P. Jalote, "Atomic Actions in Concurrent Systems," Ph. D. thesis, in preparation, Department of Computer Science, University of Illinois, Urbana, 1985.

[14] P. Jalote and R. H. Campbell, "Atomic Actions in Concurrent Systems," In: Proceedings 5th International Conference on Distributed Computing Systems., Denver, Colorado, 1985.

[15] K. H. Kim, "An Implementation of a Programmer-Transparent Scheme for Coordinating Concurrent Processes in Recovery," In: Proceedings COMSAC80, pp. 615-621, 1980.

[16] ----. "Approaches to Mechanization of the Conversation Scheme based on Monitors," *IEEE, Transactions on Software Engineering,* vol. SE-8, no. 3, pp. 189-197, May 1982.

[17] B. H. Liskov, "On Linguistic Support for Distributed Programs," *IEEE Transactions on Software Engineering,* vol. SE-8, no. 3, pp. 203-210, May 1982.

[18] B. H. Lisk[illegible] and R. Scheifler, "Guardians and actions: Linguistic Support for Robust, Distributed Programs." *ACM TOPLAS,* vol. 5, no. 3, pp. 381-404, July 1983.

[19] B. H. Liskov and A. Snyder "Exception Handling in CLU," *IEEE Transactions on Software Engineering,* vol. SE-5, no. 6, pp. 546-558, November 1979.

[20] D. B. Lomet, "Process Structuring, Synchronization, and Recovery using Atomic Actions," *SIGPLAN notices (ACM),* vol. 12, no. 2, pp. 128-137, March 1977.

[21] P. M. Merlin and B. Randell, "State Restoration in Distributed Systems," In: Digest of Papers FTCS-8: Eighth Annual International Symposium on Fault-Tolerant Computing., Toulouse, pp. 129-134, 1978.

[22] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering,* vol. SE-1, no. 2, pp. 220-232, June 1975.

[23] B. Randell, P. A. Lee and P. C. Treleaven, "Reliability Issues in Computing System Design," *ACM Computing Surveys,* vol. 10, no. 2, pp. 123-165, June 1978.

[24] D. P. Reed, "Implementing Atomic Actions on Decentralized Data," *ACM Transactions on Computer Systems,* vol. 1, no. 1, pp. 3-23, February 1983.

[25] D. L. Russell and M. J. Tiedeman, "Multiprocess Recovery using Conversations." In: Digest of Papers FTCS-9: Ninth Annual International Symposium on Fault-Tolerant Computing, Madison, WI, pp. 106-109, 1979.

[26] S. K. Shrivastava, "Concurrent Pascal with Backward Error Recovery: Implementation," *Software-Practice and Experience,* vol. 9, pp. 1021-1033, 1979.

[27] A. Silberschatz, "Communicating and Synchronization in Distributed Systems," *IEEE Transactions on Software Engineering,* vol. SE-5, no. 6, pp. 542-546, November 1979.

[28] W. G. Wood, "A Decentralized Recovery Control Protocol," In: Digest of Papers FTCS-11: Eleventh Annual International Symposium on Fault-Tolerant Computing., Portland, pp. 159-164, 1981.

APPENDIX C

An Implementation of Path Pascal

Dirk C. Grunwald

MS Thesis, Department of Computer Science,
University of Illinois at Urbana-Champaign,
Urbana, Illinois, 1985.

AN IMPLEMENTATION OF PATH PASCAL

BY

DIRK CLAUS GRUNWALD

B.S., University of Illinois, 1983

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1985

Urbana, Illinois

Acknowledgements

This work was done while the author was supported by research contract NASA NSG1471 provided by the National Aeronautics and Space Administration.

My thanks go to my thesis advisor, Dr. Roy Campbell for his time and advice, Ray Essick for his tolerance of annoying questions and to Song Moon and Malcolm Railey for their willingness to aid in debugging the compiler.

TABLE OF CONTENTS

CHAPTER 1.

INTRODUCTION

The computer language Path Pascal was designed to provide a facility for expressing concurrent programs while demonstrating the combined used of data encapsulation and process synchronization by the method of path expressions. The language, and a previous implementation, have been described in detail elsewhere [Campbell1].

The thrust of this thesis is to describe an implementation of the language based on a portable compiler system running under the UNIX[†] operating system and the process of implementing that compiler. The resultant compiler is rather large and a detailed explanation of the changes would prove to be rather tedious. Thus, an over-view of the algorithms and and data structures entailed in the modifications is presented rather than a compendium of module-specific modifications. This allows one to understand the methods behind the implementation as well as the reasons that guided the selection of those methods.

The second chapter provides an overview of the previous implementation as well as an overview of the Berkeley Pascal compiler on which the new implementation is based. It seeks to justify the need for a re-implementation of the language and the choice of the Berkeley Pascal compiler as the vehicle for that implementation. The third chapter covers the differences in the language implemented when compared to the original Path Pascal compiler. It also covers some issues concerning the semantics of Path Pascal in more detail than is available elsewhere. The fourth chapter documents the compiler modifications

[†] UNIX is a trademark of Bell Laboratories.

required to expand the Berkeley Pascal compiler to recognize the language extensions in Path Pascal. The fifth chapter covers the additional run-time system required for concurrent programming language extensions. The sixth chapter addresses the performance relative to the previous implementation based on experiences of the Path Pascal user community. Following this is a conclusion and a list of references.

CHAPTER 2.

Background Information

In order to provide the needed perspective for the remainder of the thesis, a review of the original implementation of Path Pascal, the Berkeley Pascal compiler and the justifications for transforming the Berkeley Pascal compiler into a Path Pascal compiler is required.

2.1. The Original Path Pascal Compiler

The first implementation of the Path Pascal language was based on the P-code Pascal compiler [Ammann], which is a recursive-descent, single pass compiler generating a symbolic assembly language targeted for a stack machine, called the Stack Computer, designed to be particularly suitable for executing Pascal programs.

The extensions to the compiler are documented in appendix E of [Kolstad], as are the extensions to the P-code produced by that compiler. Since the output of the compiler was designed for a hypothetical stack machine, the generated code was either interpreted or translated into assembly language for a specific machine. This compiler provided no facility for separate compilation of programs, nor did it provide an easy mechanism for interfacing the programs with code written in other languages. Another shortcoming was an inability to map abstract Pascal data types to machine specific representations, a feature which is very important for systems-programming languages.

In addition to this, writing a machine-specific translator is a difficult and time-consuming task. While implementations were provided for the VAX family of processors as well as the Motorola 68000 family, these implementations were never proven to be sufficiently robust for general use. A robust implementation for the PDP-11 family was produced and was used to implement an experimental operating system.

The language was most often used for simulating concurrent systems, particularly in class room situations. This required a stable execution environment which was provided by an Extended P-code interpreter written in Pascal. While this solution was acceptable for small simulations, it limited the possibilities for the language.

2.2. The Berkeley Pascal Compiler

As part of their efforts to expand the usefulness of the UNIX operating system, a Pascal compiler was written for the Berkeley 4.1 BSD operating system at the University of California, Berkeley. The compiler is a multi-pass compiler based on a LALR(1) grammar generated using the YACC compiler-compiler [Johnson1].

Rather than re-write the compiler for specific machine architectures, the compiler was written to produce Portable C Compiler Intermediate Representation, or PCCIR [Kesseler]. The output from the compiler is directed to a machine-specific pass of the compiler which generates native assembly code. The code-generation phase is shared between a variety of compilers running under the UNIX system, reducing the amount of work needed to port the system as a whole [Johnson2], [Joy1], [Joy2].

This compiler provided a reasonable separate compilation facility and appeared to be portable across a variety of machine architectures. In addition, it provided a richer

mapping of software-to-hardware data representations than that provided by the old compiler. It also defined a common procedure calling mechanism across a variety of languages. This allows routines to be written in other languages when appropriate.

2.3. Synthesis

In an effort to provide a better environment for the Path Pascal language, it was decided to expand the Berkeley Pascal compiler to provide the features of the P-code Path Pascal compiler. This involved changes to the compiler itself as well as the supporting runtime system. The effort was designed to provide as much compatibility with the previous implementation as possible while exploiting the features of the Berkeley Pascal environment. Although the original target for the compiler was the VAX-11 family of machines, it was intended that sufficient generality should be maintained within the compiler to allow it to be ported to a variety of machines.

Since a major use of the Path Pascal language is simulation of discrete event systems, faithfulness to the original implementation was to be stressed; however, since one of the goals of a new compiler for Path Pascal was to provide a basis for constructing operating systems and promoting future research in distributed systems implementation, the implementation was designed to keep in mind future extensions and additions to the language.

CHAPTER 3.

Language Differences

While an effort was made to maintain compatibility with the previous implementation of Path Pascal, slight differences have been introduced due to the nature of the Berkeley Pascal compiler and through experience with the previous compiler. Most differences are related to the fact that a different, more robust base compiler is used. Details concerning the basic Pascal implementation such as the size of sets, storage allocated for variables, additional pre-defined procedures and functions are detailed in the Berkeley Pascal Users Manual [Joy1]. This section of the thesis concerns itself with differences between the old compiler and the new compiler which would be evident when porting a Path Pascal program written using the old compiler. In addition, extensions to the basic Path Pascal language which have been implemented in the new compiler are also noted.

Details concerning the exact syntax of the language implemented is illustrated in the grammar given in Appendix A and will not be discussed in this section

3.1. Differences Related to Objects

A central feature of the Path Pascal language is the object notation for data encapsulation. This facility provides a means of abstracting and hiding transforms on data while maintaining an 'arena' for concurrent activity, similar to the monitor construct of other languages [Hoare], [Lampson].

Some changes have been made to the syntax and semantics of the object data encapsulation notation. Most of these changes are syntactic in nature and introduce little incompatibility with the previous implementation. While the changes are to be relatively minor, it can be seen from the language grammar in Appendix A that they introduce subtle differences in the language which cause require some consideration when using the new compiler for programs developed using the previous implementation.

3.1.1. Path Expression Notation

The first compiler provided Open Path Expressions [Campbell2] as the concurrency control method within objects, allowing the programmer to specify constraints on concurrent execution. Open Path Expressions provide three constraints on entry points to objects: resource restriction, sequencing and resource derestriction. In addition, a list building comma allows several entry names to be grouped together. Thus, a path expression such as:

path 1:(a,b.c) **end**;

causes the restriction operation '1:' to be applied to all elements of the list '(a,b,c)'.

The original implementation allowed multiple references to a single entry routine within the same path-list, with the execution constraints imposed by the synchronization operations enforced in a left-to-right fashion.

This means that the comma was no longer a non-associative operation designed to distribute operations over several entries. Typically, path expressions were written using the comma in two different senses. The first use was as a list building constructor, and the

second use was to connect separate path expressions.

As an example of this, consider the Open Path Expression used to specify the synchronization for a shared-pointer bounded-buffer queue. In essence, two restrictions need to be imposed by the path expression. The first restriction states that no more items should be removed from the queue than were originally placed on the queue, or in the notation of [Campbell3] :

$$0 \leq S(\text{enqueue}) - S(\text{dequeue}) \leq \text{bound}$$

This can simply be expressed by the path expression:

path bound:(enqueue;dequeue) **end**;

The second restriction states that no instances of dequeue and enqueue should be active at the same time, or restated:

$$0 \leq S(\text{enqueue}) + S(\text{dequeue}) \leq 1$$

Let us assume that this must be imposed because a shared pointer is used to point to the head of the queue. This restriction is simply:

path 1:(enqueue,dequeue) **end**;

Using the notation of the original compiler, these two restrictions would be written using a single path expression, as in:

path bound:(enqueue;dequeue), 1:(enqueue,dequeue) **end**

The comma operation now has two semantic meanings from the programmers viewpoint.

In one sense, it connects the two separate path expressions together, and in the other sense, it provides a list constructor. It was felt that this was potentially confusing, and the syntax was changed to allow multiple path expressions while enforcing a single reference to a entry routine within a given path expression. This separates the two functions of the comma, reducing it to a simple list constructor. Thus, the preceding example would be restated as

```
path bound:(enqueue;dequeue) end;
path 1:(enqueue,dequeue) end;
```

using the new path expression syntax. The multiple path expressions are enforced sequentially.

3.1.2. Initially Procedure

Each object may specify an optional initialization block that is to be executed each time an instance of that object is created.

The new compiler treats the initialization procedure as a standard procedure with a different syntax for the procedure declaration. All standard Pascal constructs are available within the initialization procedure, and sub-procedures, variables and objects may all be declared within the initialization procedure.

Before the code in the initialization block is executed, all the semaphores within the object are initialized and the initialization procedures of all enclosed objects are called. The initialization procedure has been restricted to appear after all the other code bodies in the object declaration but before the optional 'finally' declaration, and it may not be declared to be either **forward** or **extern**. The name of the initialization procedure has been changed from **init** to **initially** to highlight these differences.

3.1.3. Export types

One facility which is not provided in the current compiler is the notion of being able to export a type or constant from within an object declaration. The structure of the compiler does not forbid its introduction; however, since the feature is little used it was omitted in favor of simplifying the implementation.

3.2. Other Syntactic Differences

While most differences in the compiler are related to the object notation, some additional differences are present. Most of these are relatively minor, with a few notable exceptions.

3.2.1. Time vs. Wallclock

In the original Path Pascal compiler, a pre-defined function named 'time' was provided to return the current time. The exact semantics of the function change when a program is run in simulated time or non-simulated (real time) mode.

In the Berkeley Pascal compiler, the name 'time' was already used as a pre-defined procedure which returns the alpha-numeric representation of the current time of day. A function with similar semantics to the Path Pascal 'time', called 'wallclock', is available. The semantics of the 'wallclock' function now correspond to the function provided by the previous 'time' function. The 'time' function of Berkeley Pascal remains unchanged.

This introduces two-way incompatibility. Previous Path Pascal programs must change the function 'time' to 'wallclock'. This can be done rather easily by defining the function shown in figure 1.

```
function time : integer;
begin
    time := wallclock;
end (* time *);
```

Figure 1

The other incompatibility introduced by this change has to do with the semantics of the 'wallclock' function for programs written in Berkeley Pascal. The semantics are defined by the run-time system, which currently only supports a notion of 'simulated time'. Thus, a program written in Berkeley Pascal using the 'wallclock' function will not function correctly when compiled using the new Path Pascal compiler. The Berkeley 'wallclock' procedure reads the current time using the 'gettimeofday' UNIX call. Since programs written in Berkeley Pascal may call subroutines written in the C programming language, it is possible to directly use the UNIX system call.

3.2.2. Type Equivalence

The original Path Pascal compiler used a notion of structure equivalence for determining compatible type declarations. The Berkeley Pascal compiler uses the notion of name equivalent types. This means that the assignment in figure 2 raises a compile-time error in the new compiler but would be accepted by the original compiler. The problems raised in this example can be fixed by rewriting the procedure as shown in figure 3.

This represents a difference between the two implementations which can require significant modification of an existing program when compiling it using the new compiler.

```
procedure typeproblem;
var
    a : ^ something;
    b : ^ something;
begin
    a := b;
end;
```

Figure 2

```
procedure typeproblem;
type
    somethingptr = ^ something;
var
    a : somethingptr;
    b : somethingptr;
begin
    a := b;
end;
```

Figure 3

The use of 'anonymous types' is most often seen in declarations of pointer variables such as shown in figure 2. This difference exists due to the rather vague manner in which the original Pascal language definition [Jensen] defined type-compatibility and the choices made by the implementers of the Pascal compilers used to implement Path Pascal. In the language report, the phrase 'identical type' is used when considering assignment compatibility of types. However, no definition of the phrase 'identical type' is given. Since the proposed ISO Pascal standard [Addyman] uses name equivalent type compatibility and since name equivalent compatibility is a subset of structure equivalent compatibility, the use of name compatible types appears to be a reasonable choice for the Berkeley compiler.

3.2.3. Interrupt Processes

The old compiler provided an explicit syntax for specifying interrupt vector numbers for interrupt processes as well as a method for indicating that a process was an interrupt process. This has been omitted from the new implementation due to a desire to implement the interrupt process concept in a different form.

3.2.4. Octal Address Specification

The old compiler provided a notation for specifying absolute variable addresses in octal notation. This extra notation has been eliminated since it duplicates a function provided by the Berkeley Pascal compiler. Octal addresses are now specified using a trailing 'b', as documented in the Berkeley Pascal Users Manual.

3.3. Differences in Run-Time Semantics

Several differences are more properly evident as changes in the run-time system as opposed to the compiler and affect the semantics of the language more than the syntax.

3.3.1. Lifetime of Activation Records

Standard block-structured languages use a concept of activation records to provide storage for instances of variables declared in code bodies. For standard programming languages which have a single thread-of-control, such as Pascal, the dependency of code-bodies on any given activation record is such that it can be implemented using a stack. This can be illustrated using a pictorial representation for activation records and the static and dynamic links used to reference variables in those activation records. For example,

consider the program segment in figure 4 and the corresponding activation record dependency graph shown in figure 5.

In general, the dependency graph for sequential programs is a simple list of activation records. In an actual system, this list is implemented using a stack from which the activation records are allocated. In the diagrams given here, dashed vectors indicate a procedure or process call. Solid arcs represent claims made by an instance of a process or procedure on the activation record of another procedure. Solid boxes represent an instance of a procedure, and dashed boxes represent instances of processes.

```
procedure a;
    procedure b;
    begin end;

    procedure c;
    begin b end;
begin c end;
```

Figure 4

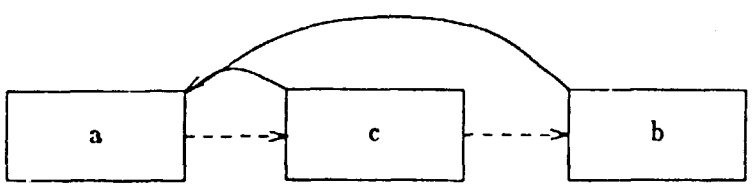


Figure 5

In languages such as Path Pascal, which may have multiple threads of control in a shared-memory environment, the data-dependency rules become complicated. This situation arises when process declarations are nested within other code bodies allowing those processes to reference items in the activation records of the enclosing code bodies. A sample code fragment illustrating this problem is presented in figure 6.

In this example, an instance of the process labeled 'one' is spawned when the process executing procedure 'illustrate' reaches the first line of that procedure. At the point of the call to procedure 'two' within procedure 'illustrate', the dependency graph is as shown in figure 7.

```
procedure illustrate;

var
    i : integer;

process one;
    begin
        delay(20);
        writeln('the value of i is ', i);
    end;

procedure two;
    begin
        delay(10);
        writeln('the value of i is ', i);
    end;

begin one; i := 10; two end
```

Figure 6

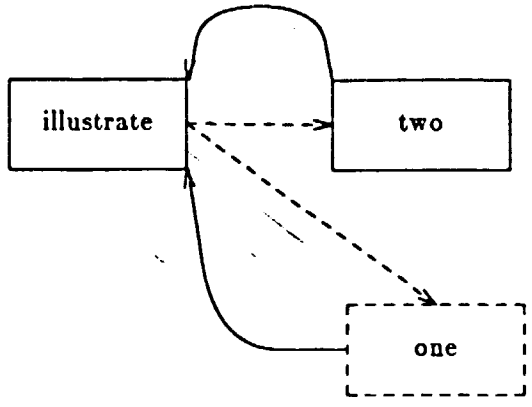


Figure 7

As indicated, the activation record of procedure 'illustrate' is shared by two processes: the process which called procedure 'illustrate' and the instance of process 'one' enclosed within the procedure. In general, the activation record dependency graph becomes a general tree with additions and deletions being made only at the leaves of the tree. Each activation record is owned by the process which entered the code body creating the activation record, and that process is responsible for disposing of the activation record.

In general, the activation records could be allocated from a heap and be manipulated as a general graph. Typically, however, greater efficiency is gained by allocating a stack to every process and imposing restrictions on the possible parallelism in the program.

Considering the example above, we see that the process executing procedure 'illustrate' must preserve the activation record of that procedure until process 'one' completes. In a system using a global heap and a garbage collector, this could be done by a simple

reference count in activation records and a garbage collection phase. This is a large price to pay for a concurrent programming language. Since Path Pascal programs are similar in structure to Pascal programs. they are typically decomposed into subroutines and subroutine calls are very common. These subroutine calls and calls to standard procedures, such as 'writeln' and 'new', allocate activation records. Most of these subroutines do not include nested processes, and most activation records are not shared by multiple processes. Forcing a rather expensive procedure calling convention for every procedure call would make the language unattractive for system implementation. In addition, it limits the ability to use subroutines written in other languages since they would probably not use the same procedure calling conventions.

In an implementation using a single stack per process, the process executing procedure 'illustrate' would need to wait for the process 'one' to complete and then free the space for the activation record from its stack. This method, called 'wait-for-sons', was used in the original Path Pascal system and is also the approach used in the new compiler.

While the two systems use the same implementation to solve the problem, the semantics of the two constructs are slightly different. In the original compiler, no use-definition information was maintained in the symbol table for variables. This forced the conservative approach of having every code body maintain a count of the number of processes referencing the activation record for that code body. This count is inspected in the procedure post-amble and the process suspends itself if there are any outstanding references to the activation record. The last child to de-reference the activation record is responsible for restarting the parent process.

This runtime check, which is performed for every procedure call, represents another penalty for procedure calls beyond maintaining the standard dynamic and static links that Pascal systems typically use. This overhead is avoided in the new compiler by making use of use-definition information already maintained by the Berkeley Pascal compiler as well as keeping track of other information related to nested processes. If a procedure does not contain a process declaration or process call, there is no possibility of sharing activation records. Likewise, if a process declaration is nested within a procedure, and that process does not reference any variables within the enclosing procedure(s), there is no reason to force a wait. Problems introduced by referencing possibly aliased variables, declared using the **var** parameter passing mechanism, are avoided by including the procedure parameters within the activation record. A reference to procedure parameters by a nested process causes the process which owns the activation record for that procedure to suspend execution until the nested process terminates.

The wait-for-sons operation provides an additional synchronization mechanism beyond the path expressions provided by the object mechanism. It is possible that some programs use this fact to synchronize processes, and thus, there may be some slight differences in program execution when comparing the new compiler to the old compiler. This is, in fact, the case, as this problem was noticed in an incorrectly coded example in the compiler validation suite.

While this introduces a substantial semantic difference across the language implementations, the original language definition did not specify details concerning the wait-for-sons synchronization, and the method was purposefully left open-ended for a variety of implementation possibilities. In a true shared memory multi-processor system, some form of gar-

bage collected activation record management might prove useful to allow maximal concurrent activity while in a uni-processor, the wait-for-sons method would be preferred for efficiency. Thus, it could be argued that the programs using the wait-for-sons synchronization are incorrectly coded. This appears to be a valid claim since the Open Path Expressions were intended to be the only synchronization construct in the language.

An example program which illustrates this difference is shown in figure 8. The dependency graph for this program is given in figure 9.

In the old interpreter system, the process labeled 'c' would complete execution before the main process was allowed to free the activation record for procedure 'b'. Under the new system, process 'c' depends only on the activation record for procedure 'a', and thus

```
program test(output);

procedure a;
var k : integer;

    procedure b;
    var j : integer;

        process c;
        begin
            writeln('c starts'); k := 10;
        end;

    begin c end;

begin b; writeln('return from b') end;

begin a end.
```

Figure 8

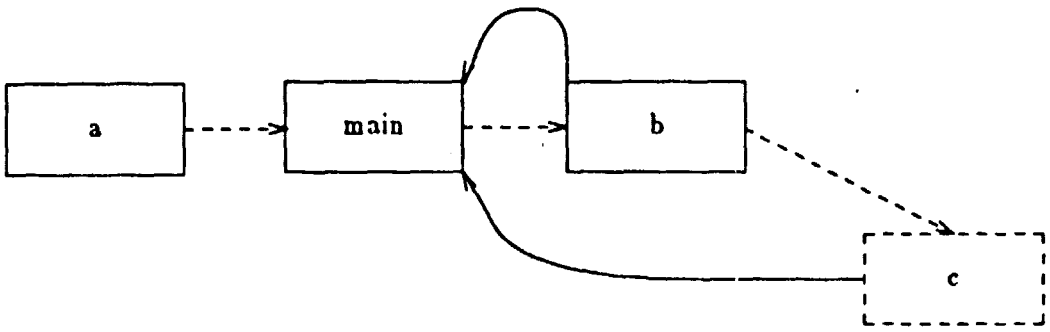


Figure 9

the main process does not wait on the exit of procedure 'b', but must wait before exiting procedure 'a'. Using the old compiler, the results of this program would be:

```
c starts
return from b
```

Using the new compiler, the output would be:

```
return from b
c starts
```

The delay in executing process 'c' arises because the main process does not need to wait before exiting procedure 'b' since process 'c' does not reference the activation record of procedure 'b'.

There is another way to cause multiple references to an activation record which is not detected or handled by either the current Path Pascal compiler or the previous compiler. This can arise when a variable within a procedure is passed to a process using call-by-address parameter passing (**var** parameters). References such as these can be found during

compilation, but the process which references the activation record can not easily determine the activation record on which it depends. This requires a more sophisticated process spawning method, and since it was not supported by the original compiler, it was decided to forego implementing it in the new compiler until alternate solutions could be considered. A sample program which illustrates this problem is shown in figure 10.

To simplify several other implementation problems concerning shared global resources such as files, all processes have an implicit dependency on the main process, which is not allowed to terminate until all other processes have finished.

3.4. Extensions to the Language

An extension to the object notation, external objects, has been provided to facilitate separate compilation at the object level. The syntax is illustrated in Appendix A and file

```
program sample(output);

    process a(var ref : integer);
    begin
        delay(10); ref := 10;
    end;

    procedure invokeerror;
    var
        i : integer;

    begin a(i); end;

begin invokeerror end.
```

Figure 10

naming conventions will be described in detail in the Path Pascal User Manual.

An optional procedure automatically executed whenever an instance of an object is destroyed as been added to provide a matching construct for the 'initially' procedure. The 'finally' procedure appears as the last code body in the object declaration, using the keyword **finally** to label the code body. In a manner similar to the 'initially' procedure, the 'finally' procedures of any objects enclosed in the current object are executed after the 'finally' procedure for the current object.

Other extensions are the result of using the Berkeley Pascal compiler as the base Pascal compiler and are documented in the Berkeley Pascal Users Manual [Joy1].

CHAPTER 4.

Modifications to the Compiler

Adding the extensions required for the Path Pascal language required several changes to the Berkeley Pascal compiler, although no new major data structures were needed and the structure of the Path Pascal compiler is very similar to the original Pascal compiler. This section of the thesis seeks to provide an overview of the data structures used in the compiler and what modifications were needed to implement the major extensions to the language in the order they were implemented.

4.1. Extant Data Structures

Before describing the changes made to the compiler, it will be useful to review the data structures used in the Berkeley Pascal compiler. The data structures in the compiler can be broken down into two major categories: those related to parse tree representation and those related to symbol table management.

4.1.1. Parse Tree Representation

The parse tree is represented by a simple list data structure which assumes that the compiler writer can determine the structure of the parse tree by looking at the first node in the sub-tree of interest. The storage for the parse tree is allocated in a stack fashion following the static nesting levels of the program. Storage for the parse-tree representing a procedure or function is released once that procedure or function is parsed. No changes were made to this data structure other than the addition of several new parse tree nodes to

represent the new productions added to the language grammar.

4.1.2. Hash Table

The Hash Table is a key element of the symbol table. It provides a one-to-one mapping of character strings to integers. The integer returned by the mapping is actually a pointer to a single copy of the character string. This allows all string comparisons to be done using a simple integer comparison, and the hashing mechanism plays an important part in simplifying the design of the Name Table and the Display Table. Once a string is entered into the Hash Table, it is never removed. The mapping from strings to Hash Table entries is done during the lexical analysis phase of the compiler.

4.1.3. Name List

The Name List is the principal data structure in the symbol table. Entries in the Name List are used to maintain Pascal type information as well as information binding names to procedures, functions, variables, types and constants. Each Name List entry has a type associated with it indicating the use of that Name List entry. There are additional fields which have a standard meaning across all Name List types. These maintain information about the entry such as storage class, resolved forward declarations, use and modification information, whether the item contains an instance of a file, the block level at which the entry was created, and an optional name associated with the entry and linking information to maintain a linked-list structure associated with the entry. The entries are typically linked together to allow access to all information related to a specific language unit by referencing a certain node in the Name List. As an example, a record definition node maintains links to all the fields of the record definition. Thus, the record definition

node is the only node which must be 'remembered' by the compiler, and the links can be used to retrieve the actual information concerning subfields within the record.

While the uses of the various fields were somewhat obscurely documented in the Berkeley compiler, the documentation has been extensively revised to illustrate the use of the various fields since this data structure is central to the compiler.

Items added to the Name List within a procedure are not automatically removed at the end of that procedure. Instead, a 'reset mark' is maintained to indicate the range of items to be removed from the name table. This alleviates the problem of maintaining a free list of Name List nodes and cleaning up discarded entry nodes. The Name List nodes must exist as long as any item with them is referenced within the compiler. Since Pascal uses a nesting, or stack oriented, visibility rule, the Name List entries associated with the names declared within a procedure or function can be freed at the end of that procedure or function body since those names will not longer be needed. The Name List does not govern the allocation of storage within an activation record for a code body. This is handled by a separate machine-specific module of the compiler which determines the sizes and alignments of data items.

4.1.4. Display Table

The compiler maintains a Display Table to keep track of names which are currently available according to the name visibility rules of Pascal. Only those names contained in the Display Table at a given point may be referenced by the program being compiled. When a code body is entered, all names related to programmer declared symbols declared in that code body are added to the Display Table. There are several types of 'hidden'

Name List entries which are never added to the Display Table and are solely used to maintain internal compiler information. The Display Table is kept as a 64-way bucket hash based on the name field of the Name List entries added to the Display Table.

When a code body is completed, all names in that code body are removed from the Display Table. At that time, the use and modification information of the entries being removed is checked and diagnostic information may be printed. This Display Table should not be confused with the run-time display table which is used to maintain the static nesting links for between activation records during program execution. The latter is the most common use of the word 'display', and while the compiler Display Table is closely associated with the runtime 'display', it performs a separate function.

4.2. Processes

Processes were the first Path Pascal extension to be added to the new compiler, principally because they are syntactically similar to procedures and the semantics are fairly easy to implement. Their implementation provided an exercise in familiarity with the Berkeley Pascal compiler, providing an ability to estimate the time and extent of modifications which would be required to implement the entire Path Pascal language.

The major implementation decisions to be made centered around the method used to 'spawn' a process. In the original Path Pascal compiler, the code to spawn a process was generated at the point of each process call. Since the new compiler needed to support separate compilation of processes, this was not a possible option. Instead, a process call appears to have the same form as a procedure call at the assembly language level. Control is temporarily transferred to the called process which then calls the run-time system to

actually allocate and create the process. At that point, control returns to the process caller just as if a procedure call had been done. This method was not used in the original Path Pascal compiler due to the argument passing and procedure calling conventions used by the Stack Computer.

This method has the advantage of localizing all information needed in spawning the process, such as the process size. This allows a simplified separate compilation interface where the process-specific parameters need only be specified in the file where the process is actually defined. This method also required very few changes to the compiler data structures.

The data structure which was modified was the Name List. A new Name List node type was created to represent processes. The source code was changed to treat the 'process' nodes exactly like 'procedure' nodes except in the actual code-generation subroutines. In those routines, a different code body pre-amble and post-amble is generated for processes.

In addition to this, other data structures were added. An array called 'hasprocess' was added to indicate whether a static nesting level contains a process declaration. The array 'maxdeplevel' was added to record the maximum static nesting level (and thus the activation record at that nesting level) on which the current nesting level depends. This information is used to implement the 'wait-for-sons' operation in the procedure post-amble code.

4.3. Objects

The implementation of the object construct was the most difficult aspect of the entire compiler implementation. Objects affect the visibility rules of Pascal, hiding information or, in the case of entry procedures, making certain pieces of information visible.

As with processes, the implementation details of objects guided the required changes to the compiler. Since it was desired to provide a separate compilation facility for objects, decisions affecting the dispersal of information concerning the implementation details of objects are critical in the design of the compiler. Local variables in the Berkeley Pascal compiler are bound relative to the current activation record at compile time. The number of bytes needed to represent an object must be known at compile time to allow the compiler to allocate sufficient room for instances of the object. To avoid over-specifying the implementation of a separately compiled object, a simple implementation interface is provided, separating the object into an implementation and an implementation interface. This hides implementation issues when using separately compiled objects and allows the programmer to focus on the correct use of the object and not the implementation. The implementation interface lists the entry procedures, functions and processes. All other information concerning the object is specified in the actual object implementation.

This implementation interface does not provide enough information to the compiler to allow it to allocate memory for the object. Instead of requiring more information to be specified for separately compiled objects, objects were implemented using an indirect heap-based method. The 'size' of an object is the size of a pointer to the item in the heap. This pointer contains the address of the actual data area for the object. The actual storage for the object is declared by the initialization procedure for the object, which is also responsible for initializing the semaphores for the object.

This method introduces some penalty for using objects. The heap management software is reasonably efficient, although not as efficient as a simple stack discipline. The cost associated with the additional pointer dereference is insignificant. Since the only way

data may be accessed by procedures external to the object definition is via a procedure call, the additional indirection is a minimal addition to the existing overhead.

For data accesses within the object definition, the indirection cost would appear to be significant. However, by treating the object definition boundary as a static nesting level and using the run-time display table to maintain pointers to the storage area for the object, much as procedures use the display for pointers to activation records, the extra indirection needed to reference the object on the heap is eliminated. This method has the additional advantage of requiring little modification to the structure of the compiler since the code to allocate variables within a static nesting level and to load and store data relative to pointers stored in the display table is already written. Since the program stack grows from high memory down to low memory, and objects are referenced from a base address which is the lowest address in the object, this method does require that the storage allocation routines be changed to allocate variables from low memory to high memory if they are being allocated for an object and from high memory to low if they are being allocated for a normal activation record.

The implementation requires that the address for the data area for an object be passed to the entry procedure on each procedure call. This address is entered into the run-time display table. Consider the sample program shown in figure 11. Within entry procedure 'a' the run-time display table would be as shown in figure 12. In this diagram, the dotted box represents the storage for the instance of the object named 'x'. Thus, a call to an entry procedure in an object appears to be preceded by an 'invisible procedure call' which establishes the environment surrounding the entry procedure and then calls the entry procedure. The code to set up the surrounding environment is actually in the pro-

```
program showdisplay;

var
    j : integer;
    x : object
        path a end;
        var
            i : integer;

        entry procedure a;
        begin i := j; end;

        end (* object *);
begin j := 10; x.a; end.
```

Figure 11

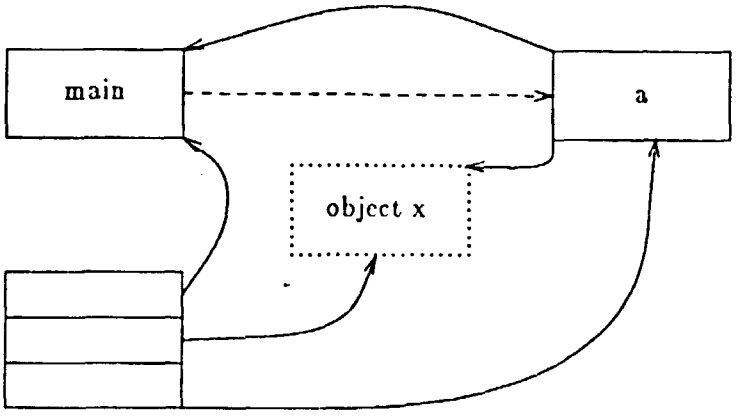


Figure 12

cedure pre-amble of the entry procedure.

Not only does using an indirect object instantiation method provide for flexible separate compilation, it leaves the door open for alternative object access methods. When the language is expanded to include the Distributed Path Pascal 'remote object' extensions, the implementation should prove easier due to this implementation decision.

One problem associated with implementing objects is providing names for the code bodies associated with the object. Most of the work associated with separate compilation is done by the program linker. A unique naming must be established for the code bodies associated with each object in the final program to allow the linker to link together the various code segments correctly. The use of 'anonymous types', that is, un-named types used in a 'var' definition of an object, are a problem because multiple names may be associated with the same object. The choice of which name to use to label the code bodies affects the usefulness of separate compilation since a consistent naming scheme is needed. To remove the problem in the initial implementation, separate compilation of objects is restricted to objects declared in 'type' declarations, as is specified in the production labeled 'type_ext_obj_item' in the grammar in Appendix A. The type name of an object provides a unique naming for code bodies associated with the object.

Each object contains two standard procedures which are not defined by the programmer. The first procedure, called '$init', allocates the data area for the object, initializes the semaphore variables, calls the '$init' routine for any nested object instantiations and then calls the 'initially' procedure if one was specified. This routine is called whenever an object is instantiated. Similarly, there is a procedure called '$fini' which first calls the 'finially' procedure if one was specified, then calls the '$fini' routine for any nested object instantiations and and then deallocates the object data area.

Implementing the 'entry' procedure concept required changes to the Display Table, because of the changes in the visibility rules due to the addition of objects. Within an object, the entry procedures and the definition for the object itself must be visible. Entry procedures become visible once they are defined. The name entering routines for the Display Table were changed to allow inserting names at any static nesting level as opposed to only the current nesting level. This is used to 'push out' the definition of entry names into the nesting level containing the object. This change also allows 'forward' declarations of entry procedures as shown in figure 13.

```
program woof(output);

type
   xptr = ^x;

   x = object

      path a, b end;

      var next : xptr;

      entry procedure b; forward;

      entry procedure a;
      begin new(next); next^.b; end;

      entry procedure b;
      begin new(next); next^.a; end;

      end (* object *);

begin x^.a; end.
```

Figure 13

The Name List data structure was also changed to not dispose of the name list nodes declared within the scope of the object definition. These nodes must be saved for the duration of the object visibility within the program because the parameter lists, entry definitions and other information concerning the object is stored in the static nesting level of the object. Since the Name List is normally allocated and de-allocated according to the static nesting level, this information would be lost without this change. Consider object 'x' in figure 13. The name 'x' is recorded in the static nesting level of the main program, and the Name List nodes of all the entry procedures are allocated within the static nesting level of object 'x'. If these nodes were deleted at the end of object 'x', the nodes for procedures 'a' and 'b' would be removed. This means the entry names would not be visible outside the scope of the object.

The data structures used to implement the 'wait-for-sons' synchronization were also changed. Names referenced within an object definition actually cause a reference to the activation record enclosing the object definition and not to the nesting level of the object itself. Thus, a data structure was added to record whether we are nested within an object definition and the record keeping routines for the 'wait-for-son' semantics were changed to use this table when updating activation record reference information since an object does not represent a nesting level which actually declares an activation record. However, within the compiler, the object appears to be an actual nesting level caused by a procedure definition. References to the 'activation record of the nesting level', which are actually references to the storage for the object, should actually increment the reference count of the procedure containing the object. In figure 13, references to variable 'next' in procedures 'b' and 'c' actually cause the reference count of the activation record for the main program to be incremented.

4.3.1. Path Expressions

Path expressions were implemented after the object notation had been installed. The original algorithm for translating Open Path Expressions was for a top-down recursive descent style compiler. This algorithm was changed somewhat for the LALR(1)-based Berkeley compiler.

The algorithm scans the parse tree of the path expression and generates a set of Name List nodes describing the required series of semaphore operations. This list is attached to the Name List entry for a given object and is searched when generating the procedure pre- and post-ambles for entry procedures. Each procedure maintains a list of operations on semaphores and pointers to those semaphores (which are also entries in the Name List). Each procedure has a separate list for the pre- and post-amble semaphore lists. An outline for the algorithm appears in figure 14.

The subroutine 'AddSemOp' performs most of the work. The parameter 'Kind' specifies whether this is a pre- or post-amble synchronization, the parameter 'ParseTree' points to the Parse Tree, the parameter 'Sem' points to the semaphore to add, the parameter 'Op' specifies the type of semaphore operation to add (P-OP, V-OP, PP-OP or VV-OP) and the parameter 'PathOp' specifies which kind of path operation (RESTRICTION, SEQUENCE, DE-RESTRICTION) is being added. The algorithm for making additions to the list is very straight forward and is essentially the top-down algorithm using some knowledge concerning the productions in the YACC grammar. The subroutines 'PathSequence', 'PathRestriction' and 'PathDeRestriction' are called when the productions 'path_seq', 'path_res' and 'path_deres' in the grammar (see Appendix A) are performed. The operations of later path expressions involving the same procedure are simply added to

```
LeftMost(X) :-
        return pointer to left-most ITEM or LIST in parse-tree X

RightMost(X) :-
        return rightmost ITEM or LIST in parse tree X

AddSemOp(Kind ParseTree, Sem, Op, PathOp) :-
{
        if (ParseTree = ITEM) {
                if (PathOp = RESTRICTION)
                        if (Kind = PRE)
                                add Sem to tail of PRE-list for procedure
                        else
                                add Sem to tail of POST-list for procedure
                else
                        if (Kind = PRE)
                                add Sem to tail of PRE-list for procedure
                        else
                                add Sem to tail of POST-list for procedure
        } else
        if (ParseTree = LIST) {
                recurse and distribute the operation over all the
                elements within the LIST, using the same SEM,
                Kind, Op and PathOp values.
        } else
        if (ParseTree = SEQUENCE) {
                if (Kind = PRE)
                        AddSemOp( leftmost( ParseTree ) )
                else
                        AddSemOp( rightmost( ParseTree ) )
        }
}

PathSequence(ParseTree) :-
        s = NewSemaphore
        AddSemOp( POST, rightmost(ParseTree), s, V-OP, SEQUENCE)
        AddSemOp( PRE, leftmost(ParseTree), s, P-OP, SEQUENCE)

PathRestriction(ParseTree) :-
        s = NewSemaphore
        AddSemOp( PRE, ParseTree, s, P-OP, RESTRICTION)
        AddSemOp( POST, ParseTree, s, V-OP, RESTRICTION)

PathDeRestriction(ParseTree) :-
        s = NewSemaphore
        AddSemOp( PRE, ParseTree, s, PP-OP, DERESTRICTION)
        AddSemOp( POST, ParseTree, s, VV-OP, DERESTRICTION)
```

Figure 14
Algorithm for Constructing Semaphore Lists

C-2

the end of the list, simplifying the implementation of the multiple path expressions in the new compiler.

This algorithm constructs the list of semaphore operations. The algorithm to interpret this list and generate the appropriate synchronization operations is presented in figure 15.

The code to actually perform the various semaphore operations calls standard procedures in the run-time library. This allows a variety of semaphore implementations to be

```
GenSynch(Direction, S-List) :-
{
        case S-List.kind {
                P-OP, V-OP:
                        if (Direction = POST)
                                gensynch(POST, S-List.next)
                        generate simple semaphore operation
                        if (Direction = PRE)
                                gensynch(PRE, S-List.next)

                PP-OP, VV-OP:
                        generate P-OP to lock counting semaphore
                        gensynch( dire, S-list.next)
                        generate V-OP to unlock counting semaphore
        }
}

GenPreAmble( Procedure )
        S-list = Procedure.SemaphoreList
        NewS-List = reverse(S-list)
        GenSynch(PRE, NewS-List)

GenPostAmble( Procedure )
        S-list = Procedure.SemaphoreList
        NewS-List = reverse(S-list)
        GenSynch(Post, NewS-List)
```

Figure 15
Generation of Sempahore Operations

used, maintaining compiler portability. In practice, the semaphore operations are actually performed by in-line code which is added by a separate pass of the compiler designed to optimize calls to oft-called standard subroutines.

While no formal proof of equivalence between the two Open Path Expression translating algorithms has been produced at this time, the similarity of the algorithms would suggest that they are indeed equivalent. Practice has shown no differences in synchronization when programs have been ported from the old compiler to the new.

4.4. Size of the Additions

The original Berkeley Pascal compiler used a common set of source files to implement a program formatter, the compiler and an interpreter. These files were approximately 21,500 lines of code written in the C programming language. The language grammar was specified using the YACC compiler-compiler, and the YACC grammar file was about 900 lines long. It was decided that the Path Pascal system would not be using the interpreter software, so the additional source used to implement the interpreter and program formatter was removed to simplify the compiler. This resulted in a base compiler size of about 18,500 lines. The extensions for processes added approximately 300 lines of code, those for objects, 1400 lines of code and those for path expressions, 800 lines. The total Path Pascal compiler size is now about 21,500 lines long, with the YACC grammar being 1200 lines long. These lengths reflect both lines of code and documentation, although they do not reflect documentation added to other areas of the compiler.

CHAPTER 5.

The Run-time Environment

Most computer languages require a run-time environment to provide support for implementation specific features, such as file operations, memory allocation as well as support for semantics defined in the language such as run-time type checking, set manipulation and so on. The run-time environment for the Berkeley Pascal compiler was structured to use the run-time libraries designed for the programming language C layered on top of the standard UNIX buffered I/O library and the standard UNIX math functions library.

The Berkeley Pascal system provides a library containing routines which enforce the Pascal semantics for file I/O, perform operations on set variables and array variables and support the run-time semantics of the case control structure and subrange checking for subrange types. In addition to this, several run-time data structures are maintained, including the display used to access variables according to the static nesting level of procedures. Some of the details concerning the Berkeley run-time library can be found in [Joy2].

The semantics of the Path Pascal language requires a much more extensive set of run-time routines to support a multiprogramming environment as well as operations on semaphores. It was decided that these routines should be written to take advantage of the existing run-time library. This set of routines, called the Path Pascal Kernel, is structured as a layer between the Path Pascal program and the host operating system. The operations discussed in this chapter provide the minimal Kernel which supports the 'simulated time' semantics of the Path Pascal system on the VAX family of computers running the UNIX operating system. The simulated time scheduler does not provide for pre-emptive

scheduling of processes, simplifying many implementation issues. While a system involving pre-emptive scheduling and interrupt processes is currently being constructed for the VAX/UNIX environment, the details of this implementation would only serve to obfuscate rather than clarify the structure of the Kernel.

5.1. Context Switching

The central elements of the Path Pascal Kernel are the context-switching routines and the process management subroutines. The relationship between the Kernel and the Path Pascal program is shown in figure 16. The Kernel runs as a co-process with the user processes. Service requests for the Kernel are directed through the context switching mechanism. From the viewpoint of both the Kernel and the user processes, the context switch appears to be a simple procedure call.

From the viewpoint of the kernel, an entry point called 'rts_exec' is available to execute a process. That process executes until it requests an operation from the Kernel. At that point, it pushes the operands for the request and an operation identifier on the runtime stack and then calls the subroutine 'rts_call'. The subroutines 'rts_exec' and 'rts_call' perform the context switch in an implementation-specific manner. It is within these routines that all process-specific data structures need to be saved. In the Berkeley Pascal environment, the only data structure which needs to be saved and restored is the runtime display, although other data structures used within the Path Pascal Kernel need to be saved over context switches.

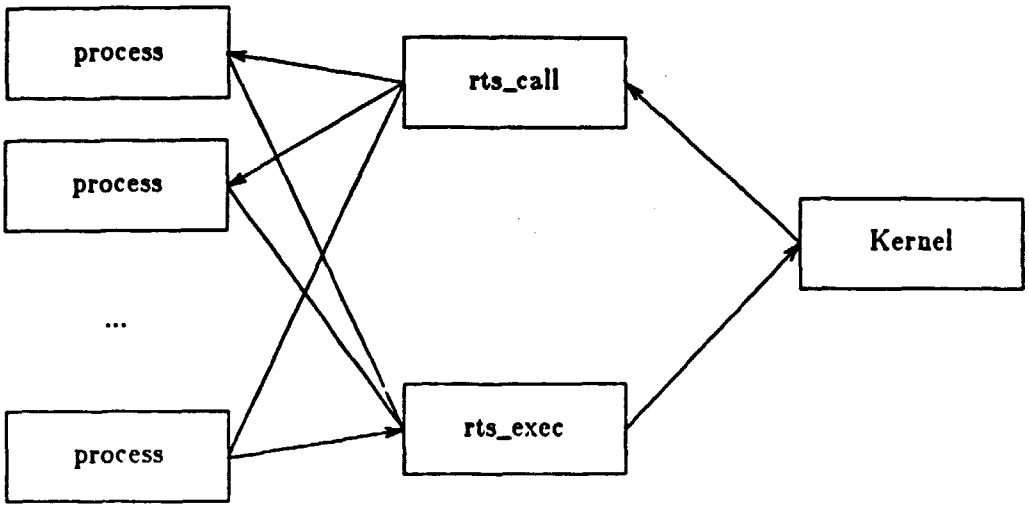


Figure 16

5.2. Kernel Overview

Within the Kernel, a process is represented by a 'process control block' which maintains all the information concerning the process, including the information needed to perform a context switch. Other information stored is used to determine which queue the process is waiting on, what the status of the process is and what area of memory has been allotted to the process.

A process is in one of seven states:

- Done
- Ready
- Executing
- Suspended
- Delayed
- Blocked on a Semaphore

These states are closely associated with the various queues used within the Kernel to manage the processes. The 'ready queue' is the queue of all processes ready to execute. The 'delay queue' is the queue of all processes waiting for a time event. The 'suspended queue' is used to keep track of suspended processes.

The state of the process is stored in a variable within the process control block for that process. When a process is marked 'Done', it has finished executing and is waiting to be reclaimed. Typically, processes do not remain in this state very long since processes are immediately reclaimed. A process is 'Ready' when it has been entered on the ready queue but is not currently executing. When a process is executing, its state is set to 'Executing'. A 'Suspended' process is one which is waiting for its last child to awaken it before exiting a procedure which contains a shared activation record. A process is 'Delayed' if it is waiting for a time event to occur. When a process attempts a P-operation on a semaphore and fails, its state is set to 'Blocked on a Semaphore'. This state information is maintained within the Kernel when a process makes a state transition and is used to guard against potential corruption of any of the queue data structures. If a bad process transition is detected, an error message is raised and a diagnostic dump of the program is generated. The allowed process state transitions are illustrated in Figure 17.

5.3. Kernel Operations

The operations performed by the Kernel can be broken down into operations involving process transitions, operations involving semaphores and other services.

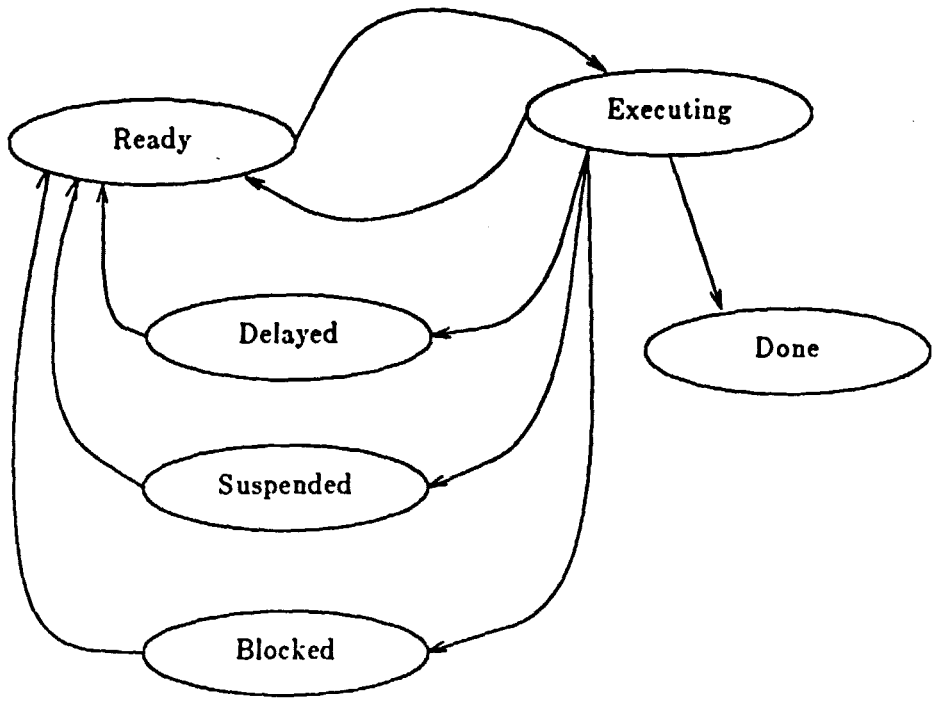


Figure 17

5.3.1. Process Operations

The process transition operations allow one to

- Allocate a process
- Free a process
- Wait-For-Sons

The Kernel must maintain the 'wait-for-sons' semantics, which requires additional state information for each process. Each process has a recorded 'dependency level' which indicates the static nesting level of the inner-most activation record on which the process depends. Space is allocated in each activation record to record the count of current refer-

ences. When a process is created, the reference count of the dependent activation record is incremented, a process control block and process storage space is allotted for the process and the parameter stack is constructed so as to appear to be created by a procedure call. The process is initialized to a state which starts its execution at the first instruction of the process code.

When a process reaches the end of the process body, it does not perform a normal end-of-procedure return sequence. Instead, it requests that the Kernel dispose of the process storage space and process control block. At the same time, the process decrements the reference count on the dependent activation record. If that count returns to zero and the parent process is suspended, the parent process is restarted. Each process maintains a pointer to the process which created it. Each process also maintains a 'process display', which is somewhat similar to the static nesting level display used in a typical Pascal implementation. The 'process display' is used to keep track of the pointers to the process control blocks of the processes which own the activation records pointed to by the run-time display table. A copy of this is saved at context switching time in the process control block, much as is done with the display table.

When a process enters a procedure where there is some possibility of multiple references to the activation record, the count field in the activation record is cleared and the pointer to the process control block of the current process is entered into the process display. When a process reaches the end of a code body where it may be necessary to check for dependencies on the current activation record, the count field in the activation record is checked. If it is non-zero, the process issues a 'suspend process' request to the kernel, expecting to be awakened by its last child process.

5.3.2. Semaphore Operations

While there are typically only two operations defined on semaphores, P and V, efficiency concerns in the Kernel dictate that four operations be implemented: P, V, Enqueue and Dequeue. The reasoning behind this has to do with the frequency of semaphore operations and the cost associated with a Kernel operation. The semaphore operations P and V have the normal semantics and perform the entire operation within the Kernel.

In implementations where it is feasible, the P and V operations are expanded by a third pass of the compiler, generating in-line code to process the P and V operation. This expanded code uses the the Enqueue and Dequeue operations of the Kernel to actually move processes on and off of queues. These queue manipulation operations must be performed within the Kernel because they need to appear as atomic operations to the external world. In systems implementing interrupt processing, the Kernel typically disables interrupt processing while it is executing to ensure that this is the case.

The data structure used to implement semaphores is shared by both the compiler and the run-time system. Extensive modifications to the data structure would require modifications to the semaphore initialization and allocation routines of the path expression compilation section of the compiler. Fortunately, it is not envisioned that the data structure will be changed often.

5.3.3. Other Services

The only other services provided by the Kernel at this time are the system initialization and delay queue management subroutines. The system initialization is called by the

program pre-amble and is used to initialize the process control blocks, statistics counters and to other data structures used by the Kernel.

The delay management services provides a time ordered event-list. Processes can request to either 'delay' for some specified number of time units or they may 'await' a certain time. They are placed on the delay queue until the specified time is reached at which time they are once again placed on the ready queue.

5.4. Kernel Size and Performance

One of the original goals of the Kernel was to provide a low-overhead implementation allowing the Path Pascal compiler to be used in many applications for which the Berkeley Pascal compiler would typically be used.

This goal has been met by a conservative and parsimonious design which stressed modularity, well defined interfaces and simple semantics. The additional cost for using the Path Pascal compiler for a program which does not use the Path Pascal extensions is minimal, allowing a single compiler to be used for both Pascal and Path Pascal programs.

Several changes in the Kernel implementation arose due to rather unexpected performance problems. One factor which had been overlooked was the affect of a paging memory on Kernel data structures. The process control blocks for processes are scattered through memory, being allotted and disposed as processes are created and destroyed. When adding a process to a semaphore or queue, the Kernel traversed the list of control blocks associated with the semaphore or queue in order to maintain a priority queue. In simulations involving several hundred processes, it was found that this involved not only a considerable amount of traversal time, it also caused every page in the UNIX process to be touched,

causing a tremendous number of page faults. This caused the access methods for the data structures to be redesigned to avoid this problem. Another unexpected cost was for the actual allocation and release of process control blocks. In programs which create many short-lived processes, this cost quickly becomes high enough to merit additional Kernel code to fix the problem. To this end, a 'PCB pool' of process control blocks is kept. This reduced the average run time of sample process-bound programs by approximately five to ten percent.

In the VAX-11 implementation, the Kernel is implemented in about 50 lines of assembly language (for the context switching routines) and 900 lines of code written in the C programming language. Another 100 lines of definitions and constants is also present. The VAX Kernel implements the simple 'simulated time' semantics in which the value returned by 'wallclock' does not advance if processes are ready to run. If there are no ready processes, the system clock is advanced to the time specified by the process at the head of the delay queue. That process is then ready to execute. The statistics for the source code given above includes an extensive amount of debugging information which has been left in the Kernel to make future modification easier. Should that code be removed, the size of the Kernel could be reduced by approximately twenty percent.

CHAPTER 6.

Compatibility and Performance

This chapter seeks to address compatibility and performance gains from the perspective of the compiler user. These measures are rather subjective in some cases, although it is felt that they reflect the actual change in performance that one can expect as well as the time required to port programs from the old compiler to the new.

6.1. Comparison to the Old Compiler

Compatibility must be judged from two directions: from using the old compiler to using the new compiler and from using Berkeley Pascal to using the new compiler. The former is to be expected due to the nature of the compiler, and the latter is required as a goal of the compiler. The language differences in the versions of Path Pascal implemented have been detailed in chapter three. Rather than reiterate those points, the significance of those changes from the viewpoint of the compiler user is presented. In a series of programs which were changed to run under the new compiler, the editing changes required to port most of the programs entailed approximately five minutes of modifications. In cases where extensive use of the 'structure compatible' type declarations had been made, the time to port the program was increased, although the use of a good text editor simplifyed the process immensely. As an example, a 1300 line program which is used to simulate a file-server running on an Ethernet was modified to use the new compiler. Marginal use of the structure equivalent typing was made. The total time needed to modify the program for it to compile correctly was under five minutes. The time would have been somewhat more

lengthy if the procedure 'time', as shown in figure 1, had not been used to redirect calls from procedure 'time' to procedure 'wallclock'.

The time to compile and run the program, as well as the size of the resultant output file is shown in figure 18. The size of the output measured reflects the completely linked UNIX load file for the new compiler and the Pcode file for the old compiler. The compiler size information is in the format 'text_pages + data_pages', where each page is one kilobyte long. Since the new compiler is broken into several small passes, the size of each piece is smaller than the old compiler. This is useful for small systems with a limited address space. However there is more paging activity caused by this structuring since the separate programs needed to be loaded sequentially.

The compile time for the new compiler is approximately three times that of the old compiler. The principal reason for this is that the compiler involves many passes over different files, whereas the old compiler is a true one pass compiler generating textual pseudo-code. The first pass of the new compiler, which is the main pass which was changed, requires only 15.8 CPU seconds to run. Any errors encountered in the program would be detected by the first pass, and compilation would stop shortly thereafter. Thus, in a classroom environment where many more compiles than executions are performed because of the many errors common in student programs, the new compiler would probably out-perform the old compiler. Also, one must realize that the new compiler performs

Version	Compile Time (sec)	Compiler Size(Kbytes)	Object Size (Kbytes)	Run Time (secs)
New	92.4	89+140	15.8	48
Old	34.0	336+132	929.9	75

Figure 18

much more work, preparing the program for immediate loading and execution. The old compiler delays some of these expenses until the interpreting stage. Another factor to consider is that the program compiled was developed using the old compiler. Typically, in an environment which supports separate compilation, such a program would be broken into several smaller files allowing a quicker compile-debug development cycle.

The program run times presented for the old compiler system are, to a certain extent, grossly exaggerated because the older system uses an interpreter to run the program instead of using actual compiled code. However, one should be concerned with extant tools, not illusorily ones. In the current environment, the interpreter represents the best run times possible for programs developed using the old compiler. The difference in runtimes between the Berkeley Pascal interpreter and the Berkeley Pascal compiler is reported as five to fifteen times faster execution when using the compiler. When a series of measurements were taken on the difference between interpreted and compiled Path Pascal, the speed-up was seen to be lie in the range of a 40 to 150 times increase. This is a sizable disparity, and the chief reason it is so great is that the Path Pascal interpreter is written in Pascal and was the major design emphasis was portability rather than speed. The Berkeley interpreter was designed for speed above portability, and the compiler produces execution rates varying from 5 to 15 times greater than their interpreter rates [Joy2].

One other factor which should be considered when comparing the usability of the new compiler to the old system is the increased range of problems which can be tackled and the savings in human time. As an example, one set of programs used in the simulation of concurrency control methods in distributed data base management systems [Moon] required run times of over 8000 CPU seconds using the old compiler and interpreter. Due to the

unreliability of the hardware and the use of the machine for other development efforts, these programs would often not complete execution before the machine was rebooted. Using the new compiler, these run times were reduced to 50 seconds, allowing the project to be completed with much less work.

In other programs involving simulation of several hundred processes to measure the effectiveness of distributed control algorithms [Railey], the problem could not have been solved using the old compiler and interpreter due to the large amount of storage and processes required. The old system had defined limits on the size of simulated memory, where as the new compiler uses whatever memory is available. On a virtual-memory machine, this means several hundred processes may be simulated, something not possible under the old system.

6.2. Comparison to Berkeley Pascal

Another factor to consider with the new compiler is the added overhead of using the Path Pascal compiler instead of the Berkeley Pascal compiler. The results in figure 19 are compilation and run times for 'pascref', a 750 line Pascal cross-reference program originally written by Niklaus Wirth. The program was then used to produce a cross-reference listing of itself.

Version	Compile Time (sec)	Compiler Size(Kbytes)	Object Size (Kbytes)	Run Time (secs)
Berkeley Pascal	76.3	81+124	685	32.1
Path Pascal	78.6	87+133	679	32.9

Figure 19

The difference in compilation and execution time demonstrates that the Kernel design was such as to reduce overhead on programs which do not use the Path Pascal extensions. The slight increase in execution time can be attributed to the additional start-up and shut-down costs incurred by using the Path Pascal Kernel. The compilation time increase is mainly due to the larger language grammar, requiring more time to be spent in the parser, the more complicated code associated with the end of procedures and the additional link time needed to load the Kernel. The differences in the size of the load image can mainly be attributed to the Kernel and its associated data structures.

As can be seen, the use of the Path Pascal compiler causes a slight decrease in performance when compared to the Berkeley Pascal compiler. However, the cost is rather minimal, and may be reduced by 'tuning' the compiler.

CHAPTER 7.

Conclusion

As has been shown, the design goals of the compiler have largely been met. The compiler provides a significant execution performance increase over the existing system at the expense of slightly longer compile times.

The language implemented is largely compatible with the previous version, requiring only minimal program changes to port programs from the old compiler to the new one. Furthermore, the language is a direct superset of the Berkeley Pascal language (with the exception of the 'wallclock' standard procedure), allowing a single compiler to be used as the sole development tool. The development environment supported by the compiler is much more flexible than that supported by the original Path Pascal compiler.

The compiler, Kernel and post-mortem debugging tools, users manual and automated installation procedures were implemented in approximately ten man-months. A pre-release of the system has been distributed to about fifteen UNIX sites around the country and in Korea.

The compiler was originally targeted for the VAX family of machines, and has also been ported to the SUN workstations using the Motorola 68000 CPU. Further ports are expected to other 68000-based machines as well as machines based on other CPU architectures.

APPENDIX A.

Path Pascal Language Grammar

The following is the list of reserved words in the Path Pascal language:

and	array	begin	case	const	div
do	downto else	end	entry	extern	
file	finally	for	forward	function	goto
hex	if	in	initially	interrupt	label
mod	nil	not	object	oct	of
or	others	packed	path	type	procedure
process	progam	record	repeat	set	then
to	until	var	while	with	

The following defines the precedence relations for operators in expressions in YACC notation. These are used by the YACC compiler-compiler to eliminate shift-reduce conflicts in the grammar. See [Johnson1] for detailed information.

%binary	<	=	>	in	
%left	+	-	or		
%left	- (unary)				
%left	*	/	div	mod	
%left	not				

The following is the grammar for the language. The grammar is left in a form similar to that which is given to the YACC compiler-compiler to illustrate the sequences of

productions. Several apparently useless productions are performed, but are the order of reductions is required in the structure of the compiler for passing information between different productions.

```
goal:
    prog_hedr decls block '.' | decls ;

prog_hedr:
    program ID '(' id_list ')' ';' | program ID ';' |
    program error ;          /* See note Error */

block:
    begin stat_list end ;

decls:
    decls decl | decls error
    | /* lambda */          /* See note Lambda */ ;

decl:
    labels | const_decl | type_decl | var_decl | proc_decl ;

labels:
    label label_decl ';' ;

label_decl:
    INT | label_decl ',' INT ;

const_decl:
    const ID '=' const ';' | const_decl ID '=' const ';' |
    const error | const_decl error ;

type_decl:
    type_ytype type_item | type_decl type_item |
    type_ytype error | type_decl error ;

type_ytype:
    type ;

type_item:
    ID '=' type_noobjects ';'
        |
    type_obj_item decls init_fini end ';'
        |
    type_ext_obj_item ext_entry_decls end ';' ;
```

```
type_obj_item:
    type_obj_item_kludge path_expr ;          /* See note Force */

type_obj_item_kludge:
    ID '=' object ;

type_ext_obj_item:
    ID '=' extern object ;

ext_entry_decls:
    ext_entry_decls phead_tree | /* lambda */ ;


var_decl:
    var var_id_list ':' type ';' | var_decl var_id_list ':' type ';' |
    var error | var_decl error ;


var_id_list:
    vid | var_id_list ',' vid ;


vid:
    ID '[' number ']' | ID ;


proc_decl:
    phead forward ';' | phead extern ';' |
    pheadres decls block ';' | phead error ;


pheadres:                    /* See Note Force2 */
    phead ;

phead:
    phead_tree ;


phead_tree:
    isentry porf ID params ftype ';' |
    isentry process ID sizepart params ftype ';' ;

isentry:
    entry | /* lambda */ ;


porf:
    procedure | function ;


sizepart:
    '[' number ']' | /* lambda */ ;
```

```
params:
    '(' param_list ')' | /* lambda */ ;


param:
    var id_list ':' type |
    function id_list params ftype | procedure id_list params ftype |
    id_list ':' type | process id_list params ;


ftype:
    ':' type | /* lambda */ ;


param_list:
    param | param_list ';' param ;


const:
    STRING | number | '+' number | '-' number ;


number:
    const_id | INT | BINT | NUMB ;


const_list:
    const | const_list ',' const ;


type:
    type_noobjects | object ;


type_noobjects:
    simple_type | '^' ID | struct_type | packed struct_type ;


simple_type:
    type_id | '(' id_list ')' | const .. const ;


struct_type:
    array '[' simple_type_list ']' of type |
    file of type | set of simple_type | record field_list end ;


simple_type_list:
    simple_type | simple_type_list ',' simple_type ;

field_list:
    fixed_part variant_part ;
```

```
fixed_part:
    field | fixed_part ';' field | fixed_part error ;

field:
    /* lambda */ | id_list ':' type ;

variant_part:
    /* lambda */ | case type_id of variant_list |
    case ID ':' type_id of variant_list ;

variant_list:
    variant | variant_list ';' variant | variant_list error ;

variant:
    /* lambda */ | const_list ':' '(' field_list ')' ;

object:
    obj_path decls init_fini end ;

obj_path:
    object_hdr path_expr ;

object_hdr:
    object ;

path_expr:
    path_single | path_expr path_single ;

path_single:
    path_hdr path_list end ';' | error ;

path_hdr:
    path ;

path_list:
    path_seq | path_list ',' path_seq ;

path_seq:
    path_item | path_seq ';' path_item ;

path_item:
    ID | '(' path_list ')' | '[' path_list ']' | number ':' path_item ;
```

```
    init_fini:
        initially finally          /* See note Reduce */
            |
        initially | finally | /* lambda */ ;


    initially:
        initial_hdr decls block ';' ;

    initial_hdr:
        initially ';' ;

    finally:
        final_hdr decls block ';' ;

    final_hdr:
        finally ';' ;


    stat_list:
        stat | stat_lsth stat ;



    stat_lsth:
        stat_list ';' ;


    cstat_list:
        cstat | cstat_list ';' cstat | error | cstat_list error ;


    cstat:
        const_list ':' stat | otherwise ':' stat | /* lambda */ ;

stat:
    /* lambda */
        |
    INT ':' stat |
    proc_or_pros_id |
    proc_or_pros_id '(' wexpr_list ')' |
    ID error |
    assign |
    qual_var | qual_var '(' wexpr_list ')' |
    begin stat_list end |
    case expr of cstat_list end |
    with var_list do stat |
    while expr do stat |
    repeat stat_list until expr |
    for assign to expr do stat |
    for assign downto expr do stat |
    goto INT |
    if expr then stat | if expr then stat else stat |
```

```
    error ;

    assign:
        variable ':' '=' expr ;

expr:
    error
        |
    expr relop expr              %prec '<' \ /* See note Prec */
        |
    '+' expr                  %prec UNARYSIGN
        |
    '-' expr                  %prec UNARYSIGN
        |
    expr addop expr                %prec '+'
        |
    expr divop expr                %prec '*'
        |
    nil | STRING | INT | BINT | NUMB | variable | ID error |
    func_id '(' wexpr_list ')' | qual_var '(' wexpr_list ')' | '(' expr ')' |
    negop expr                 %prec not
        |
    '[' element_list ']' | '[' ']' ;

    element_list:
        element | element_list ',' element ;
    element:
        expr | expr .. expr ;

    variable:
        ID | qual_var ;

    qual_var:
        array_id '[' expr_list ']' | qual_var '[' expr_list ']' |
        record_id '.' field_id | qual_var '.' field_id |
        ptr_id '^' | qual_var '^' ;

    wexpr:
        expr | expr ':' expr | expr ':' expr ':' expr | expr octhex | expr ':' expr octhex ;

    octhex:
        oct | hex ;

    expr_list:
        expr | expr_list ',' expr ;
```

```
wexpr_list:
    wexpr | wexpr_list ',' wexpr ;


relop:
    '=' | '<' | '>' | '<' '>' | '<' '=' | '>' '=' | in ;


addop:
    '+' | '-' | or | '|' ;


divop:
    '*' | '/' | div | mod | and | '&' ;


negop:
    not | '~' ;


var_list:
    variable | var_list ',' variable ;


id_list:
    ID | id_list ',' ID ;


const_id:                    /* See note ConstId */
    ID ;
type_id:
    ID ;
var_id:
    ID ;
array_id:
    ID ;
ptr_id:
    ID ;
record_id:
    ID ;
field_id:
    ID ;
func_id:
    ID ;
proc_or_pros_id:
    ID ;
```

Notes:

Error

The token 'error' is used by YACC and the error recovery techniques of the compiler to produce a 'correct' parse when an error is encountered. Proper placement of 'error' tokens allows the parse to (possibly) bypass errors and continue. See [Johnson1].

Lamba

Productions marked */* lamba */* are nil-productions indicating the possibility of an empty reduction.

Force

As mentioned, some productions look odd because the compiler needs to know certain information at certain times. In this example, the keyword object must be encountered before the path expression is parsed because the code for the path expressions 'hangs' information concerning the path expressions on the Name List node allotted for the object. The Name List for the object needs a name associated with it, and in this case, the name is the associated type name. Thus, the productions force the name and the object keyword to be reduced at the same time and before the rest of the object, allowing us to set up data structures.

Force2

This is another instance of an apparently useless reduction. This example is a side-effect of the structure of the existing compiler at the time it was being extended. Rather than make major modifications in order to achieve some affect, it was easier to force small useless reductions which would cause some associated action to occur at that time.

Reduce

This production has this form to avoid reduce-reduce conflicts. Consider the productions

```
object  : ... other tokens ... init_fini end
init_fini:
        initially finally ;
initially :
        /* lambda */ | initially .... ;
finally :
        /* lambda */ | finally .... ;
```

The string "end" can either reduce to an initially or a finally in this context. By listing the productions as was done, this is avoided by not having two lambda productions possibly preceding the end.

Prec

The %prec notation is used by the YACC compiler-compiler to eliminate possible shift-reduce conflicts by assigning priorities to productions. For details, see [Johnson1].

ConstId

These productions are augmented with semantic analysis routines using the symbol table of the compiler to either declare an error or to indicate that the identifier is of the appropriate identifier class.

REFERENCES

[Addyman]
Addyman, A. M. *A Draft Proposal for Pascal.* ACM SIGPLAN Vol. 15, No. 4, pp1-66, 1980.

[Ammann]
Ammann, U., Nori, K. and Jacobi, C. *The Portable Pascal Compiler.* Institut Fur Informatik, EIDG, Technische Hochschule CH-8096, Zurich.

[Campbell1]
Campbell, R.H. and Kolstad, R.B. *An Overview of Path Pascal's Design and Path Pascal User Manual.* SIGPLAN Vol. 15, No. 9, pp144, 1980.

[Campbell2]
Campbell. R.H. *Path Expressions: A Technique for Specifying Process Synchronization.* Ph.D. Thesis, University of Newcastle upon Tyne, August, 1976.

[Campbell3]
Campbell, R.H. *Distributed Path Pascal* In: Distributed Computer Systems, ed. Y. Paker and J.P. Verjus (London, Academic Press, 1983) pp191-223.

[Hoare]
Hoare, C. *Monitors: An Operating System Structuring Concept.* CACM Vol. 17, No. 10, pp549-557, 1974.

[Lampson]
Lampson, B. and Sturgis, H. *Experience with Process and Monitors in MESA.* CACM, Vol. 23, No. 2, pp105-117, 1980.

[Jensen]
Jensen, K. and Wirth, N. *Pascal User Manual and Report.* Springer-Verlag, New York, 1975.

[Johnson1]
Johnson, S. *Yacc: Yet Another Compiler Compiler.* UNIX† Programmer's Manual, Seventh Edition, Volume 2, 1980. July 1979.

[Johnson2]
Johnson, S. *A Tour Through the Portable C Compiler.* UNIX Programmer's Manual, Seventh Edition, Volume 2, 1980.

† UNIX is a trademark of Bell Laboratories.

[Joy1]
Joy, W. Graham, S. and Haley, C. *Berkeley Pascal Users Manual, Version 2.0.* Technical Report, Dept. of EECS, University of Calif, Berkeley, 1979.

[Joy2]
Joy, W. and McKusick, M. *Berkeley Pascal PX implementation Notes, Version 2.0 - January 1979.* Technical Report, Dept. of EECS, University of Calif, Berkeley, 1979.

[Kessler]
Kessler, P. *The Intermediate Representation of the Portable C Compiler as used by the Berkeley Pascal Compiler.* Technical Report, Dept. of EECS, University of Calif, Berkeley, 1983.

[Kolstad]
Kolstad, R. and Campbell R.H. *Path Pascal User Manual.* Technical Report, Dept. of CS, University of Illinois, Urbana, 1980.

[Moon]
Moon, S. C. and Belford, G. G. *Performance of Concurrency Control Methods in Distributed Database Management Systems.* Proc. ISMM Intl Symp., June, 1984.

[Railey]
Railey, M. and Liu, J. *Single Leader Election Protocols Using Nonuniform Merits.* Technical Report UIUCDCS-R-85-1197, University of Illinois, Urbana, 1985.

APPENDIX D

Atomic Actions in Concurrent Systems

Pankaj Jalote

Ph.D. Thesis, Department of Computer Science,
University of Illinois at Urbana-Champaign,
Urbana, Illinois, 1985.

I

ATOMIC ACTIONS IN CONCURRENT SYSTEMS

by
Pankaj Jalote
M. S., Pennsylvania State University, 1982

Ph.D. Thesis, Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois, 1985

III

ACKNOWLEDGEMENTS

There are many people who have helped me psycologically, socially and technically. First and foremost I would like to thank my advisor Professor Roy H. Campbell for the helpful advise and direction he gave me. Discussions with Professor M.D. Mickunas, G. G. Belford and J. W. Liu are also appreciated.

I would also like to thank the members of the Systems Research Group and other friends, who offered help and friendship which helped maintain sanity, and created a friendly environment. I would also like to acknowledge the support from NASA grant NSG 1471.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The concept of an indivisible sequence of actions has been in use to design systems since the invention of the hardware interrupt. The possibility of interrupts requires the system designer to identify those primitive system activities that must be uninterruptable. Enable and disable interrupt operations allow the programming of indivisible or 'atomic' sequences of instruction executions that would be executed indivisibly by the hardware and exclude the undesirable side-effects that could arise as a result of a sudden change of control flow caused by an interrupt.

However, the concept of atomicity need not be limited to operations which are performed indivisibly by the hardware. The concept of atomicity is a general one and can be extended to general operations. By removing the hardware dependency of the definition, we can define an atomic action simply as an operation that appears primitive and indivisible to the activities outside the atomic action.

Supporting atomic actions in a single sequential process is relatively trivial. However, concurrent systems involving several processes are more complex than a single sequential process because of the possibility of interprocess information exchange. The need for atomicity, and the difficulty of implementing it in concurrent systems, was recognized in early operating systems and led to proposals for synchronization primitives such as semaphores[Dijkstra 65]. Since then, programming constructs involving atomic actions have appeared in the contexts of databases, operating systems and fault tolerance.

The definition of atomicity does not preclude the possibility that an atomic action has a structure of its own. Atomicity of an operation requires that the operation *appears* to be primitive and indivisible to other activities. An atomic action can have a structure of its own, though this structure cannot be visible outside the action. This permits the nesting of atomic actions and the decomposition of an atomic action into, perhaps many, sub-actions.

The concept of nested atomic actions is also fundamental to system design. The construction of "large" atomic actions from "smaller ones" is essential for designing concurrent software systems and is derived from the desirability of defining a system using a functional notation which excludes undesirable side-effects and permits hierarchical decomposition.

Though the term "atomic action" might seem to preclude any concurrent activity, this is an unnecessary restriction. The abstract definition does not imply that the atomic action consists of a single hardware primitive action, nor does it require that the atomic action be performed by a single process or on a single processor as long as it appears to be primitive and indivisible to any other activities that may occur at the level of the abstraction in which it is defined. At a more detailed level of abstraction, the different sub-activities constituting the atomic action may interleave with other activities. Certain restrictions must be imposed on this interleaving of sub-activities in order to preserve the atomicity at the higher level of abstraction.

The aim of this thesis is to show that atomicity is fundamental to programming concurrent systems and to demonstrate that many different concurrency control schemes which have appeared in many different contexts have actually the same goal: to provide a mechanism that ensures atomicity of system activities. In the recent literature great

emphasis has been placed on the database applications of atomic actions. We intend to show that the concept of atomicity is more general, provides many additional advantages, and unifies the solutions to many existing problems. In this thesis we argue that the concept of atomic actions is a concise conceptual basis for the design of concurrent systems, and discuss the many useful properties of atomic actions. It is shown that many different concurrency control requirements which appear in different distributed system applications actually have the same goal: to establish the atomicity of operations. Atomic actions are fundamental to the problem of concurrency control in databases, mutual exclusion in operating systems and provision of software fault tolerance. We propose the concept of an atomic action as a structuring mechanism that should be used in the design of distributed systems of concurrent processes.

1.1. Concurrent Systems

In this section we will define what we mean by a concurrent system. A *sequential program* specifies sequential execution of a list of statements. The execution of a sequential program is called a *sequential process.* A concurrent system is a system consisting of a number of concurrently active sequential processes.

The set of concurrent processes can be classified into three categories, namely *independent, competing* and *cooperating*[Anderson & Lee 81]. Concurrent processes are said to be *independent* if the sets of objects accessed by the processes are disjoint. Since the activity of each process must then be completely private from any other process, this case is conceptually the same as having many physically separate sequential processes.

The second category of concurrent processes removes this restrictive requirement. Concurrent processes are said to be *competing* if they share resources but no information

exchange between processes results from the sharing. There are objects in the system that are accessed by more than one process, but the shared access is solely due to the need to increase the utilization of scarce resources. The usage of the resources is private, in that no information flow between processes results from sharing the resources. Conventional multiprocessing systems support competing processes, with resources being shared between different jobs. Since there is no information flow between processes it is completely immaterial to the operation of each individual process that the resource is actually being shared. Each process gets the view that it is the sole user of the resources, and hence the processes behave as if they were independent. Thus, a set of competing processes can be treated as a set of independent processes.

The third category of concurrent processes is the most interesting of the three categories and imposes no restriction on information flow. *Cooperating processes* have shared access to objects which are used directly for inter-process communication. Communication between processes can be done either by the use of shared memory or by the use of message passing. Any problem where coordination of many processes is required must use cooperating processes. Operating systems need some system of cooperating processes to ensure proper access to the shared resources by different (competing) process. Cooperating processes have been an area of research for a long time, and many proposals have been made to support cooperating processes since the Dijkstra's seminal paper "cooperating sequential processes"[Dijkstra 65].

Cooperating processes create many problems which do not exist in single process systems. Interprocess communication is only meaningful if it can be performed in a controlled fashion, and if the programmer has the means to control the communication. Due to this, a major concern for cooperating processes is process coordination and synchroni-

zation between the actions of different process. Process synchronization has been an active research area and many mechanisms have been proposed. Some of the techniques for synchronizing processes are the semaphores[Dijkstra 65], monitors[Hoare 74], and path expressions[Campbell & Habermann 74]. Many languages have been proposed and implemented which use these mechanisms to support cooperating processes. These languages include Concurrent Pascal[Hansen 75], Path Pascal[Campbell & Kolstad 80a], Modula[Wirth 77].

Due to the need of synchronization between actions of different processes, many other problems occur. Problems of mutual exclusion, deadlock, concurrency control and data consistency arise in cooperating processes. In the remainder of this thesis we will not consider the first two categories and concentrate on cooperating systems only, and the term "concurrent systems" will be used for a system of cooperating processes. It should be pointed out that a system of sequential process is a special case of a system of concurrent processes. So, the results which are valid for concurrent processes will be valid for sequential processes, if the problem is applicable to sequential processes too.

In a system of communicating processes, the interprocess communication may be based on the use of shared variables or message passing. The two communication methodologies introduce somewhat different problems. In a shared memory system, the processes communicate by reading and writing a shared memory. The access to the shared data by different processes is controlled to maintain the consistency of the shared data. In a system using message passing for communication, the variables of different processes are not subject to concurrent access, because only one process accesses them. However, different problems arise, such as lost messages and duplicate messages.

1.2. Motivation

In this section we discuss the motivation for stu 'yi~g atomic actions in concurrent systems. Informally, an atomic action is an activity, possibly consisting of many steps performed by many different processors, that appears primitive and indivisible to any activity outside the atomic action. To other activities, an atomic action is like a primitive operation which transforms the state of the system from one state to another without having any intermediate states. An operation that is executed as an atomic action has the properties of non-interference, non-overlapping and strict sequencing.

In databases a transaction is the unit of processing. A transaction is a sequence of read and write actions on the entities of the database. If the actions of different transactions are not properly coordinated an inconsistent database may result. In the database literature, this problem is referred to as the *concurrency control problem.* What is really desired is that a transaction should appear to execute indivisibly. That is, to maintain the consistency of the database, a transaction should be an atomic action.

Serializability is often the criteria used to ascertain that proper control is being asserted on different transactions. The serializability criteria states that the net effect of executing transactions concurrently should be equivalent to executing the transactions serially in some order. This is exactly the strict sequencing property of atomic actions. If each transaction is performed atomically, the serializability criteria will be satisfied. Serializability of actions is a phenomenon which appears if all actions are atomic. So, once we have the ability to specify and support atomic actions, serializability of actions will be guaranteed.

An important concept in the area of operating systems is mutual exclusion. If two processes read and update the same shared set of data then the mutual exclusion

requirement permits only one process at a time to operate on the shared data. Mutual exclusion is required to prevent two or more processes from having interleaving accesses to shared data resulting in unpredictable results. However, mutual exclusion is not always required to preserve the consistency of the shared data. Mutual exclusion is often overly restrictive and leads to loss of concurrency[Lomet 77]. It is, however, simple to implement. What really is required is that the operations execute atomically. Mutual exclusion guarantees atomicity of the operations executing in mutual exclusion, but is overly restrictive. Atomic actions provide a more general property than mutual exclusion, and yet provide mutual exclusion where necessary.

Fault-tolerant techniques use protective redundancy to ensure that an erroneous system state does not lead to system failure. These methods aim to place the system in a state from which processing can proceed and failure can be averted. Techniques for fault tolerance are usually classified as backward or forward error recovery techniques[Anderson & Lee 81, Randell et. al. 78]. *Backward error recovery* involves backing up one or more processes to a previously checkpointed state, which is expected to be error free, and then attempting to continue further processing. In contrast, *forward error recovery* aims to identify the fault and correct the erroneous state of the system, before proceeding with normal processing.

Both of these fault tolerance techniques have four major phases[Randell et. al. 78]: error detection, damage assessment, error recovery, and fault treatment and continued system service. Atomic actions provide a convenient structure to support damage assessment and recovery. In the atomic action framework, the damage due to a fault is confined to some atomic action which contains both the fault and the detection of the error resulting from the fault. If the appropriate atomic action can be identified, forward

recovery techniques can examine the state of the atomic action to determine the cause of the error and attempt to restore the state to normal. Backward recovery assumes that any computation performed inside the atomic action is suspect and so should be discarded. Recovery mechanisms restore the initial state of the atomic action. For both kinds of recovery, atomic actions provide bounds on the damage produced by the fault. Many of the existing schemes for providing fault tolerance use atomic actions. The recovery block, and the conversation construct that have been proposed for backward recovery are in essence atomic actions.

As we can see, atomic actions provide us with many desirable properties. All of these properties have so far been studied in isolation from others. Atomic actions provide a single construct which can be used in operating systems in place of mutual exclusion, in databases to ensure atomicity of transactions, and in the area of fault-tolerance for supporting forward and backward recovery. Besides this, we expect atomic actions as a language construct to provide an useful tool for the programmer to structure, design and verify concurrent programs. We also expect atomic actions to be useful in specifying parallelism in programs, and provide proof for fault tolerant provisions.

1.3. Thesis Overview

The work done by other researchers on atomic actions is discussed in *chapter 2.* Atomic actions are informally defined, and then the different views about atomic actions are divided into three categories: Dynamically identified atomic actions, basic atomic actions, and recoverable atomic actions. The three views are then discussed. In dynamically identified atomic actions, the boundaries of the atomic actions are defined dynamically, based on the history of execution of the system. Recoverable atomic actions follow the "all or nothing" view, that is, an atomic action should either complete successfully or

should leave the system in the state it was when the action started. Basic atomic actions are the ones which are proposed and used in this thesis. The boundaries of such actions are statically defined, and recoverability is not a requirement of atomicity.

Chapter 3 contains a survey of techniques for constructing fault tolerant software. Both forward and backward recovery techniques are discussed. The recovery block scheme is used for backward recovery, while exception handling is used for forward recovery. These techniques are discussed first for sequential systems, and then for concurrent systems. The strategies for performing backward recovery in concurrent systems are divided into two classes. The methods in the first class use the communication history to determine dynamically the amount of roll back required to bring the system to a consistent state. The techniques in the second class use some language construct, similar to a recovery block in sequential systems, to define a structure within which recovery can be performed. For forward recovery in asynchronous systems, a technique based on atomic actions and exception resolution is described.

In *chapter 4* we present our formal model of atomic actions. A general definition of actions is given. Actions are specified in terms of their state transformation sequences. Atomic actions are defined in terms of restricted state transformation sequences. This definition of atomic actions is shown to have the strict sequencing property, and is shown to be compatible to other models of atomicity. Backward and forward recoverability of actions is defined for this model. We show that if an action is backward recoverable (or forward recoverable) then it is necessarily an atomic action.

In *chapter 5* we propose a notation to use atomic actions for fault tolerance in a system of communicating sequential processes. The technique provides a construct to implement atomic actions, which is used to support the different recovery schemes in a

complementary manner. A conversation-like scheme is employed for backward recovery. For forward recovery, an exception resolution scheme is used to resolve multiple exception occurrences into a single exception. Backward recovery is employed if forward recovery is unsuccessful. The construct is implemented using CSP primitives, and supports local compile-time and run-time checking of the forward and backward error recovery schemes.

In *chapter 6* we discuss the use of atomic actions in database systems. In databases, the unit of processing is a transaction, which is a sequence of read and write actions. Unrestricted concurrency among database transactions can result in an inconsistent database. It is required that a transaction execute atomically. We present a new protocol for coordinating transactions, called the Delay/Re-Read Protocol. This protocol uses a combination of preventive and corrective measures to ensure atomicity of transactions. The corrective measure employs a forward error recovery method. The preventive measure delays the Writes. Many properties of the protocol are proved and discussed and we show that the protocol is deadlock free and provides atomic execution to transactions.

Chapter 7 contains the conclusion and suggestions for future work regarding atomic actions. We discuss Many possible benefits of atomic actions, and we argue that atomic actions will be useful for program proving, program structuring, and proving correctness under exceptions.

CHAPTER 2

ATOMIC ACTIONS: RELATED WORK

Most of the techniques for structuring systems deal with the organization and sub-division of the static (or spatial) structure of the system. Typical issues addressed by these techniques are how the static structure of the system can be sub-divided, and how the components are statically organized. However, the pattern of interaction between the components of a system is also of interest. Interactions between components reflect the dynamic (or temporal) structure of the system. The concept of atomic actions can be used to structure the temporal activity of the system.

We consider the activity of a system component to be the sequence of state transitions of the component. Each transition is regarded as primitive and indivisible. A single state transition constitutes the simplest atomic action. These transitions are usually caused by hardware primitives, and the indivisibility of these actions is supported directly by the underlying hardware.

However, the concept of atomicity need not be limited to operations which are performed indivisibly by the hardware. The concept of atomicity is a general one and can be extended to general operations. By leaving out the hardware dependency of the definition, we can define an atomic action simply as an operation that appears primitive and indivisible to its environment.

Note that the definition says nothing about how the atomicity is provided. It does not require that the atomic action consist of a single primitive action, nor does it require that the atomic action be preformed by a single process or on a single processor. By this

abstract definition, an atomic action may contain many steps, and the steps may be performed on many processors. The restriction on the computation on the atomic action is that it must appear primitive and indivisible to its environment.

For our purposes, the environment of an atomic action consists of all the activities that are not contained within the atomic action. We have used the terms "primitive" and "indivisible" to define atomic actions intuitively. These two terms are actually equivalent because if an action is primitive, it is indivisible, and if an action is indivisible, it can be regarded as primitive. In this informal discussion we will use them together to convey a better feel for the notion of atomicity.

This definition of atomicity does not preclude the possibility that an atomic action has a structure of its own. Atomicity of an operation requires that the operation *appears* primitive and indivisible to its environment, but does not imply that the operation should literally be indivisible. An atomic action can have a structure of its own, though this structure cannot be visible to the environment of the action. This permits atomic actions to be nested, and allows an atomic action to be composed of many sub-actions. These sub-actions may be atomic, and may execute concurrently on different processors.

Nested atomic actions aid in modular decomposition of activities and provide a tool to organize the temporal structure of the system and support parallelism and concurrency of actions. Nested atomic actions have been proposed by many[Liskov & Scheifler 83, Davis 78, Lomet 77]. Nesting is a fundamental requirement of atomic actions.

By our intuitive definition of atomic actions, any action which appears primitive to its environment is considered atomic. However, two different meanings can be attached to the "primitiveness" of an action. According to one view, a primitive action implies that either the action completes successfully, or it should behave as if the action never

started. This definition implies some recovery measure to 'undo' a partially executed or unsuccessful action. The second view does not impose the recoverability requirement, and requires that recovery should be built using atomic actions.

There are also two views on how the boundary of an atomic action should be defined. According to one view, the boundary of an atomic action is identified dynamically by looking at the history of the computation. The second view is that the boundary of an atomic action should be specified statically, and that atomicity should be imposed at run time.

Due to differing opinions on these two basic issues, several different definitions of atomic actions have appeared in the literature. These views can be classified into three categories: dynamically identified atomic actions, basic atomic actions, and recoverable atomic actions. In the following sections we discuss these three viewpoints, and review the work of other researchers.

2.1. Dynamically Identified Atomic Actions

As mentioned above, atomic actions reflect the dynamic structure of the system, and atomicity of an activity actually depends on the execution of the activity. By definition of atomicity, it is not possible to predict the atomicity of an activity statically (or apriori) before the execution taken place, unless there are language constructs which impose restrictions at runtime on interactions between components of the system to ensure that the activity occurs atomically when it is executed.

Dynamically identified actions live up to the dynamic nature of atomic actions. There is no structure which actually restricts communication between different components in order to achieve atomicity. The dynamic techniques do not define the boun-

daries of atomic actions statically, but instead consider the history of the actual computation and the interactions that take place between the components during the computation, to determine which activities occurred atomically. Conditions on the history are specified to ascertain the atomicity of activities. Such approaches are useful for modeling and understanding atomic actions, but they do not provide the programmer with any mechanism to implement or specify atomicity. The atomicity of actions depends entirely on a particular execution, and it is possible that an activity may occur atomically in one execution but may not occur atomically in another execution. For this reason, the use of such approaches is limited in regard to system design. This approach does not provide any means for specifying atomic actions whose execution will be guaranteed to satisfy the atomicity criteria in any execution history.

Formal treatment of dynamically identified atomic actions by means of occurrence graphs was given in[Best & Randell 81, Best 80]. In the next section, we discuss their approach and results.

2.1.1. Occurrence Graph Model

The use of the occurrence graph model for atomic actions is proposed by Best and Randell[Best & Randell 81]. They give a purely dynamic atomicity criteria, using the executions of atomic actions as the basic formal objects. Occurrence graphs are used to describe computations. In this section we briefly describe the basic model and summarize some of the results given in[Best & Randell 81].

An occurrence graph is a directed graph in which the nodes are interpreted as events of the computation, and the edges are interpreted as the "conditions" holding between events. The direction of the edges indicate an ordering of events. If there is a

path from the node e to the node e', that indicates that e occurs before e', and is written as $e < e'$. The events e and e' are said to be "concurrent" if neither $e < e'$ nor $e' < e$.

For any activity that has been performed, there will be a subgraph in the occurrence graph of the system. The subgraph of the activity contains the relevant information about the execution of this activity. The sub-graph of an activity A can be "collapsed" into a single node (representing the event A) by the collapsing operation. The node obtained by collapsing the occurrence graph of an action represents the event for the occurrence of that action. The resulting node inherits all the edges of all the nodes of the collapsed sub-graph. By applying the collapsing operation to a given occurrence graph, a new occurrence graph is obtained which describes the same computation at a different level of abstraction, where events are no longer basic events, but may represent the activity of a complex action. The "granularity" of the actions which appear as events in the occurrence graph, is made "thicker" as the occurrence graph of the computation is taken to higher levels of abstraction.

An execution at a level of abstraction is a valid execution if the occurrence graph of the execution at that level of abstraction is acyclic. The basic occurrence graph (occurrence graph where events correspond to the execution of primitive operations) of any computation is always acyclic; any cycle in the basic graph would indicate an event being its own cause. At higher levels of abstraction, the occurrence graphs may not be acyclic. However, the graph obtained by collapsing the subgraphs of atomic actions should be acyclic. Because atomic actions are primitive activities, the occurrence graph containing events for atomic actions should be like a basic graph, and should be acyclic.

The difference between a valid and invalid execution is indicated by the absence or presence, respectively, of a cycle in the occurrence graph. The characteristic dynamic

property of atomicity therefore is that at all levels of abstraction, the occurrence graph of the computation is acyclic. That is, by collapsing the sub-graph(s) of atomic action(s), the graph obtained will represent a partial ordering on the events of the graph.

Interference-Freeness and Atomic Occurrences: An event e is defined to interfere with an activity A if it occurs strictly after part of A and strictly before another part of A. The activity A occurs atomically if it is not interfered with by any event in this fashion. This is the basic definition of atomic actions in the occurrence graph model. If an event e occurs after a part of A and before another part of A, this implies that A does not appear indivisible to e, which is against the intuitive notion of atomic action which we developed earlier in this chapter. This definition of atomicity depends on the execution of A and on the non-existence of events outside A which interfere with A.

Note that according to this definition, an activity A may be a part of a cycle and still be atomic. However, the presence of a cycle in an occurrence graph implies that one of the events (representing actions) in the cycle did not occur atomically. So, for checking the atomicity of a single action, the criteria of interference-freeness has to be used. However, for ensuring that all the actions at a level of abstraction occurred atomically, it suffices to show that the occurrence graph is acyclic at that level of abstraction.

Inherently Atomic Occurrences: The above definition of atomicity implies that the atomic occurrence of an activity depends not only on its internal structure, but also on its environment. However, there are activities which can be shown to be atomic just by examining the occurrence graph of the activity. Such an activity is called an "inherently atomic occurrence".

If an activity A is structured such that there is a path from every event that is an immediate predecessor of A (that is, there is an arc from the event to some event

belonging to A) to every immediate successor of A, then collapsing the graph of A will not result in a situation where there is an event outside A which interferes with A. Due to this restriction, there can be no event e such that $e < A$ and $A < e$, and so there can be no interfering activity.

An activity A is said to be an inherently atomic occurrence if there is a path from each immediate predecessor of A to every immediate successor of A. Such occurrences are also referred to as "contractions" or "two phase occurrences". A property of inherently atomic occurrences is that there always exists a line, called the 'cut', through the graph of A, such that from every immediate predecessor of A to every immediate successor of A there is a path which crosses this 'cut'.

2.2. Recoverable Atomic Actions

In this section we discuss the second category of atomic actions. In contrast to dynamically identified atomic actions, recoverable atomic actions have statically defined boundaries. That is, an action is specified to be atomic at the time the system is designed. The underlying implementation assures the actual atomicity at run-time by restricting the interactions between components during the execution. In dynamically identified actions, the execution of an atomic action never appears to overlap with the execution of another activity. The effect of failures is not considered. Recoverable atomic actions, on the other hand, uphold the "all-or-nothing" view, which requires that either all the objects changed by the atomic action change to their final state, or all of the objects remain in their initial state. This definition of atomicity is more restrictive than the definition of atomicity in basic atomic actions or dynamically identified actions.

Due to indivisibility, an atomic action appears to other actions as a state transformer that transforms the state of the system indivisibly from the initial state to

the final state of the action. The intermediate states should not be visible. Both concurrency and failure can expose the intermediate state of an action, and so, according to the view of recoverable atomic actions, concurrency and failures both threaten to violate the atomicity of an action. Hence, the atomicity requirement can be decomposed into two basic requirements[Reed 83].

(1) *Concurrency Atomicity.* For all primitive steps o which are not in the atomic action A, either o precedes all the steps in A or o follows all the steps in A.

(2) *Failure Atomicity.* Either all steps in A complete, or none of them complete.

These two requirements are referred to as indivisibility and totality, respectively, in[Allchin & McKendry 83], and Liskov refers to them as indivisibility and recoverability[Liskov & Scheifler 83]. The definitions of dynamically identified actions and basic actions consider concurrency atomicity to be the basic requirement for atomic actions, and do not consider recoverability as a requirement for atomic actions.

This definition is useful in some contexts, such as databases, where transactions, the atomic actions in databases, are required to be recoverable by the database consistency requirements. However, due to the added requirement of recoverability the problem of performing recovery has to be handled along with the problem of ensuring indivisibility. This usually complicates the issues, and the implementation becomes difficult.

Implementing recoverability requires some *stable storage* which is not corrupted by system or action failure. For any action, the state of the system at the start of the action has to be saved on this stable storage before the action can perform computation. This is necessary so that if the action fails, or if the system fails, the state of the system at the

start of the action will be safe on the stable storage. This can be used to restore the system to the state that existed at the start of the atomic action, thereby satisfying the recoverability requirement.

Some form of commit protocols are also needed due to the recoverability requirement. Usually the two phase commit protocol [Gray 78] is used. In the first phase of this protocol, all the processes taking part in the atomic action communicate their intention to commit, that is, to make their changes permanent. In the second phase, they actually make the changes permanent. In the first phase, if any process communicates its intent to *abort*, then in the second phase all the processes will abort. Data is stored on the stable storage after the first phase in order to handle the case of a system failure between the two phases; if the system fails after the first phase, this data will be available to complete the second phase when the system restarts.

Besides the complexity of the commit protocols themselves, complications occur due to nesting of atomic actions. If an action aborts, all its subactions abort. If a subaction commits, it is only a conditional commit, based on the eventual commit of the parent atomic action. If any of the enclosing actions abort then the commit of the subaction has to be revoked. This further complicates the issue of commit and requires some means to recognize the actions which can commit on their own, and the ones which can only commit conditionally. There are other limitations which come about due to the recoverability requirement. We will discuss some of them in a later section; now let us look briefly at some of the proposals for recoverable atomic actions.

2.2.1. Recoverable Atomic Actions in ARGUS

Atomic actions have been incorporated as a language feature in the programming language called ARGUS[Liskov & Scheifler 83]. ARGUS is intended to support the class of applications concerned with manipulation and preservation of long-lived, on-line, distributed data. Surviving hardware failures without loosing the distributed information is a major objective of the language design. We can understand why recoverability is considered a requirement for atomic actions in this context.

In ARGUS an activity is considered to be an attempt to examine and transform some data objects from their current (initial) states to new (final) states, with any number of intermediate states. An atomic action is an activity which is indivisible and recoverable, and may complete by *committing* or *aborting*. When an action aborts, the effect is as if the action had never begun; all modified objects are restored to their initial states. When an action commits, all modified objects take on their new states.

To permit recovery, some objects are classified as atomic objects or *atomic abstract data types*[Weihl & Liskov 83]. The operations on these objects have the properties of indivisibility and recoverability, that is, an operation performed on an atomic object by an action is guaranteed to be atomic. An operation on an atomic object is considered to be a subaction of the atomic action that invokes the operation. The facility of atomic objects delegates the work of ensuring atomicity of these subactions to the implementation of atomic objects, and the job of supporting atomicity is reduced to that of supporting global atomicity.

The implementation of atomic objects is based on the two phase locking model [Eswaran et. al. 76]. Two kinds of locks are permitted, read locks and write locks. All locks acquired are held until the completion of that action, to avoid the problem of

cascading aborts. The two phase commit protocol is used to ensure that the entire action either commits or the entire action aborts.

Nesting is permitted. An action may contain any number of subactions, some of which may be performed sequentially and some concurrently. The nested structure of an action cannot be observed from outside the action. Nesting is used to support concurrency between subactions of the same and different actions. Subactions appear atomic to other subactions of the same parent. Subactions commit and abort independently, and a subaction can abort without forcing its parent action to abort. However, a commit of a subaction is conditional. Even if a subaction commits, aborting its parent will abort the subaction. Actions are classified into two classes, subactions and top-level actions. The commit of top-level actions is irrevocable.

The two main concepts in ARGUS are guardians and atomic actions. Guardians are the logical nodes of the system, which maintain complete control of their local data, and provides access to the data to other guardians via handler calls. Atomic actions are the means by which distributed computation takes place in ARGUS.

2.2.2. Reed's Proposal

In[Reed 83], an implementation of atomic actions is proposed. The proposal has concentrated mainly on synchronizing simultaneous access to shared data objects. Both concurrency and failure atomicity are considered as basic for atomicity. The proposal is particularly concerned with the mechanics of implementing recoverable atomic actions in a distributed computer system, in which the nodes communicate through messages. Each node is assumed to be capable of providing stable or non-volatile storage termed "atomic stable storage".

The implementation uses "pseudotime" to order the operations on an object. The operations on data objects are classified into two categories, READ and WRITE operations. Each object is represented as an "object history" of "versions", with each WRITE on the object creating a new version. Versions are grouped together to facilitate backward recovery that is performed in case an atomic action has to be aborted. A variation of the two phase commit protocol is used to commit an action.

2.2.3. Atomic Actions in the Clouds System

The Clouds project[Allchin & McKendry 83] is studying techniques for the construction of reliable computing systems in environments of machines connected by local area networks. For reliability, the Clouds operating system uses abstract data types (objects) and recoverable atomic actions. The system has three logical entities, objects, processes, and actions. Objects are passive entities, and reside at a single node. Processes initiate all activity in the system. A process executes on a single node, and communicate only through objects. Actions are the units of work, and operation invocation on objects takes place within the context of actions.

An action may complete normally by committing, or complete abnormally by aborting. Pessimistic synchronization schemes are used to ensure the atomicity of actions. If an action has to be aborted, backw .ecovery is used to restore the state of the system. The aim of atomic actions is to implement atomic transactions, with both indivisibility and totality as basic requirements of atomicity.

2.3. Basic Atomic Actions

Basic atomic actions have similarities to both dynamically identified actions and recoverable actions. A basic atomic action is a *planned atomic action* like the recoverable

atomic action. That is, the boundary of a basic atomic action is statically defined. But, as with dynamically identified actions, indivisibility is the only requirement of atomicity. Recoverability is not considered to be a necessary part of basic atomic actions.

Many researchers have taken this view of atomic actions. Lomet studied atomic actions[Lomet 77] largely from the point of view of process structuring and synchronization. The major problems in this area, mutual exclusion, synchronization, and structuring, require only the indivisibility of operations. Hence, there is no need to impose the requirement of recoverability on atomic actions.

Anderson & Lee [Anderson & Lee 81] looked at atomic actions from the point of view of supporting damage confinement for different techniques of fault tolerance. Damage assessment and performing recovery are two different phases in supporting fault tolerance. Indivisibility is the property which is of interest for damage assessment. Requiring recoverability implies that the system employs some recovery technique over which the designer has no control. This is too restrictive and inflexible for fault tolerant applications.

Campbell and Randell[Campbell & Randell 83] needed some language construct for asynchronous systems, which could be used to support different fault tolerant techniques. They needed the flexibility to perform any recovery method yet combine the different methods, if needed. Recoverability is too restrictive a requirement for this purpose, and will not permit the kind of flexibility needed. However, if the construct has the property of indivisibility, the problem of supporting different recovery techniques is simplified. So, they too choose indivisibility as the only requirement of atomic actions.

To understand the intuitive notion of indivisibility in somewhat more concrete terms, Lomet has given a few definitions. One of them is in terms of the restrictions on

the interactions between the activities inside the atomic action and the activities outside the action. This definition is also proposed in[Anderson & Lee 81] and is used in[Campbell & Randell 83]. An atomic action is defined in terms of the absence of interactions:

> "The activity of a group of components constitutes an atomic action if there are no interactions between that group and the rest of the system for the duration of the activity."

With this restriction, all of the activity within an atomic action will appear to the rest of the system to be a single indivisible computation.

This definition does not faithfully translate the meaning of atomicity. It will guarantee indivisibility, but is overly restrictive in that there can be activities which do not satisfy the above definition but still appear indivisible to other activities. This fact is acknowledged in[Anderson & Lee 81]. We later give a more formal definition of indivisibility. Now let us look briefly at the construct proposed by Lomet to implement atomicity.

2.3.1. Lomet's Proposal

A language construct was proposed by Lomet to implement basic atomic actions[Lomet 77]. The aim was to provide a facility by which a writer of a procedure could directly state that a procedure must execute atomically. For this reason, a mechanism was proposed for writing *action procedures*. An action procedure is identified by the word *action* in the procedure header. The notation proposed for action procedures is shown below.

<identifier>: *action* (<parameter list>);

<statement list>

end;

The semantics of an action procedure are same as those of normal procedures except that action procedures are to be performed as atomic actions. Action procedures may be nested.

The effect of having action procedures is to shift the responsibility for resource acquisition and release in a way such that the atomicity of the action procedure is guaranteed, to the implementor of actions rather than the being the responsibility of the programmer using actions. Since resources that are accessed only by one process do not require any special protection, resources are syntactically divided into two classes, *shared* and *private*. The shared and private attributes apply to an object as a whole and not to its separate components. The purpose of this classification is to simplify the implementation of atomic procedures.

In order to allow conditional synchronization, some mechanism is needed to delay the entire action procedure until the test for the conditional synchronization can be satisfied. For this purpose, the *await* statement is introduced. It has the following syntax:

await (boolean expr) *then* <procedure>

The semantics of the await statement require the process to block at the await statement the Boolean condition in the statement is true. Then the procedure may be executed. Since the await statement is allowed inside an action procedure, restrictions on the Boolean expression are needed. Since the expression may be repeatedly executed until it evaluates to true, and since some activity outside the action procedure can change the variables used in the expression so that it eventually becomes true, information could flow

between activities inside the action and activities outside the action. To prevent a violation of the atomicity of the action procedure, the await statement must be primitive, and the evaluation of Boolean expression must not have any side effects.

2.4. Comparison and Discussion

The two issues on which the models differ are (a) how the boundaries of the atomic actions are defined, and (b) whether recoverability of actions is a basic requirement of atomicity. From the point of view of defining the boundaries of the atomic actions, recoverable atomic actions and basic atomic actions can be grouped into the category called *planned atomic actions.*

Planned atomic actions are atomic actions that were planned during the design of the system and are supported by some run-time mechanism. Atomic actions are identified and their boundaries are defined during the system design. Both recoverable atomic actions and basic atomic actions are planned atomic actions in that both techniques define the atomic actions statically.

This is in contrast to dynamically identified atomic actions, where atomic actions are identified dynamically by looking at the execution history. There is no language construct for atomic actions, and atomicity of activities depends entirely on the particular execution under consideration. These methods do not provide the programmer with any mechanism to specify atomic actions, and so are not very useful from the point of view of designing systems.

In planned atomic actions, we may or may not have recoverability as a basic requirement for atomicity. As we have seen, recoverability introduces many complications. It requires some stable storage and the action has to record data on the stable

storage at certain times before it can proceed with its computation. Commit protocols are needed, and methods are needed to retrieve and store data on the stable storage. Committing becomes more complicated for nested actions.

To support proper committing of nested actions, atomic actions have to be divided into two classes: top level actions, and subactions (using Liskov's terminology). This prohibits uniform treatment of all the atomic actions and requires proper classification of the atomic actions at the design time. Without recoverability all actions are similar in nature and so nested and top level actions can be handled uniformly in a similar fashion.

Recoverable actions are not suitable for programming fault tolerance. They imply that some recovery mechanism is employed by the system to ensure recoverability. This mechanism is fixed, and the designer has no control over it. So, different fault tolerant techniques cannot be programmed. Moreover, for fault tolerance, some means have to be provided to ensure continued service, after recovery is performed. In recoverable actions backward recovery is usually performed, but it does not permit the programming of alternates, which is the method of providing continued service. So, even for the backward recovery, the scope of recoverable actions is limited. And, it does not permit the programming of forward recovery techniques.

Furthermore, as LeBlanc points out in[LeBlanc 84] the "all or nothing" definition is not well suited for real-time distributed systems, simply because many operations in such systems do not naturally behave in that way. Also, such atomic actions cannot be implemented for systems which have non-recoverable objects. In such systems recoverability cannot be ensured.

With all these reasons we believe that indivisibility is fundamental but recoverability is not. If recoverability is desired, it should be programmed using basic atomic

actions. By adding the recovery primitives, both kinds of recovery can easily be programmed using basic atomic actions. A recoverable action is essentially a basic action with recovery primitives which are automatically invoked. So, a recoverable atomic action can be built using basic atomic actions, though the reverse is not true. For the rest of this thesis we will use the term atomic actions to refer to basic atomic actions.

2.5. Requirements for atomic actions

Any implementation of an atomic action must satisfy certain conditions. In this section we define those requirements. These are general requirements and are independent of the type of system or the mode of interprocess communication.

1) *Well defined boundaries :* Each atomic action should have start and end boundaries, and it should have two side boundaries. By side boundaries we mean that if there is more than one process taking part in the action then the side boundaries of the atomic action separate the processes taking part in the atomic action from those which are not. The start and end boundaries might be spread over several processes. The start boundary consist of the points (maybe in different processes) which define the initial state of the system at which the atomic action begins its computation. The end boundary correspondingly specifies points which define the state of the system after the computation has been performed. Together the boundaries enclose the amount of computation which has been specified to be atomic, and which the implementation should ensure has the property of indivisibility and atomicity.

The final state and the initial state may be made available in parts, which implies that the whole of the start boundary and whole of the end boundary need not exist at a single instance of time, but may be distributed over time. However, it is necessary that

the different processes taking part in the atomic action have a consistent view of the system at the initial and final boundaries. That is, the state a process sees at the boundaries is consistent with the state other processes see at the boundaries. Without this the action cannot be a coherent entity with each process performing some activities which are a part of a single operation which the atomic action represents.

2) *Indivisibility* : This is the most fundamental requirement of atomicity. An atomic action must not exchange information from any activity outside the boundaries of the atomic action, in a manner which will violate the indivisibility of the action. To other activities, an atomic action must be like a primitive operation which transforms the state of the system from one state to another without having any intermediate states. Due to the indivisibility requirement atomic actions have the properties of non-interference, non-overlapping and strict sequencing.

The non-interference property states that for a given initial state, the final state produced by an atomic action depends only on the computation inside the atomic action, and is not affected by any activity outside the atomic action. No two atomic actions may share internal state information and this gives the property of non-overlapping states. Thus, the computation specified by an atomic action is conceptually isolated from all other concurrent computations. Similarly, an atomic action has the property of strict sequencing with respect to other atomic actions. If variables are shared between two or more atomic actions, then the values of those variables, regardless of the concurrency between actions, are determined by a strictly sequential execution of the actions taken in some permutation. It should be pointed out that these properties of atomicity can be viewed as derivatives of the indivisibility requirement of atomic actions.

3) *Nesting :* Atomic actions may be nested. Nesting permits an atomic action to be defined in terms of other nested atomic actions, allows modular refinement and structuring of atomic activities, and permits concurrent execution of atomic actions. Only strict nesting can be allowed, that is, no boundary of a nested atomic action should cross any boundary of the enclosing action.

Without nesting no concurrency is possible between different actions if they access the same shared data. Only 'independent' actions, that is, the actions which operate on different data objects, may then execute concurrently. With nesting interleaved execution of subactions of different atomic actions is possible, while still preserving atomicity of the actions to which the subactions belong. Nesting also allows concurrent execution of subactions of the same action; if two subactions are 'independent' then they may execute concurrently. This property can be employed to exploit the inherent parallelism of operations. Different rules for decomposing atomic actions may be provided to simplify designing concurrent systems, exploit parallelism, and facilitate specifying semantics of systems utilizing atomic actions.

4) *Concurrency :* Atomic actions should support concurrency. Atomicity of operations can be trivially satisfied by restricting the operations to execute sequentially. In a system where operations are executed sequentially, there is little reason to introduce the concept of atomic actions. So, we consider concurrency a basic requirement of atomic actions, and not simply a factor to be considered for efficiency reasons. Concurrent execution is made possible by nesting, and independence of actions. Atomic actions should exploit this and permit the programming of concurrency in systems.

5) *Robustness :* An implementation should be robust. We include the properties of fairness, deadlock freeness etc. under this category. This property, is a desired property

rather than a strict, basic requirement.

CHAPTER 3

RECOVERY TECHNIQUES AND FAULT-TOLERANCE

As computer systems get more complex and ubiquitous, reliability of such systems becomes more critical. Fault-tolerance techniques enhance system reliability beyond the point which can be achieved by regular software engineering methods. In this chapter we will look at the concepts and principles of fault-tolerant software. First some basic definitions and concepts relating to fault tolerance are discussed. Then some techniques for constructing fault tolerant software are described. The techniques are classified into two groups: those which are based on backward error recovery, and those which are based on forward error recovery. Techniques in both these groups are separately discussed. In each case, first the techniques for sequential systems are described, and then the proposed extensions of the approaches to concurrent systems are discussed.

3.1. Basic Concepts

The reliability of a system is a measure of the success with which the system provides the intended service. A *failure* of a system occurs when the the behavior of the system deviates from that required by its specification[Anderson & Lee 81]. An *erroneous state* of a system in an internal state which could lead to system failure[Anderson & Lee 81], and an *error* is a part of an erroneous state which constitutes a difference from a valid state[Anderson & Lee 81]. The cause of an error is a *fault*[Anderson & Lee 81] and an error is a *manifestation of a fault*[Anderson & Lee 81].

The traditional approach to achieving reliability in computing systems has been based largely on *fault avoidance*[Randell et. al. 78]. Fault avoidance is concerned with

techniques to avoid the introduction of faults during the design and construction of the system. The use of high level design strategies, proven technologies, testing and verification methodologies, are all examples of techniques used for fault avoidance.

Another approach for improving system reliability is that of fault-tolerance. Fault-tolerant techniques use protective redundancy to ensure that occurrences of erroneous states do not result in system failures. The goal of fault-tolerance is to tolerate faults in the system, such that service can be provided despite the presence of faults. By the nature of the two methods, fault avoidance and fault tolerance are actually complementary rather than competitive approaches to system reliability.

Fault-tolerant techniques can be regarded as comprising of four phases, which provide the general structure for implementing fault-tolerance[Anderson & Lee 81]. The four phases are 1) error detection, 2) damage confinement and assessment, 3) error recovery, and 4) fault treatment and continued service. The particular strategies may vary in different phases and the order in which these phases are carried out may vary, but the starting point is always error detection. There can be much interaction between the different phases. The additional components that support these various phases can be considered to constitute the 'protective redundancy'.

1) *Error detection:* In order to tolerate a fault in a system, its effects must first be detected. Faults cannot be directly detected; only their manifestations may be observed. Thus the usual starting point for fault tolerance techniques is the detection of an erroneous state, that is a state which, in the absence of any corrective actions, could have led to a system failure. The success of any fault tolerant system is critically dependent upon the effectiveness of the techniques applied for error detection.

An useful technique for error detection can be based on checking if the outputs produced by the system conform to the specification. For such a technique there are three criteria for an ideal test[Anderson & Lee 81]. 1) The checks should be based solely on the specification, and should not be influenced by the design of the system. 2) The checks should also be independent of the system with respect to its susceptibility to faults. 3) The checks should be complete, that is, should completely represent the specifications and detect all occurrences of errors on the state the checks are applied. In practice, it is not possible to construct ideal tests, due to practical and cost considerations. Often tests are applied to check the acceptability, rather than correctness, of the system output, and are like the executable assertions[Andrews 79]. Redundancy in data structures can also help detect and correct errors[Taylor et. al. 80b, Taylor et. al. 80a].

2) *Damage confinement and assessment:* Measures for error detection can hope to identify some errors, but error detection cannot guarantee that all the damage caused due to the fault will be identified. In case of an error, much more of the system state might be in error than that on which the check is applied. Because there may be a substantial delay between the erroneous transition caused by a fault and the detection of any error, the damage can spread through the system. Thus, before attempting any error recovery it may be necessary to assess the extent of the damage caused by the fault. This assessment will depend on the structure of the system, and any constraints (known or incorporated by system design) that are placed on the flow of information in the system. Atomic actions provide a useful structure for damage assessment[Anderson & Lee 81].

3) *Error recovery:* Following error detection and damage assessment, the system must recovery from the error. The aim of this phase is to eliminate errors from the system state. The techniques for error recovery aim to transform the current erroneous sys-

tem state into a well defined and error free system state from where normal computation can begin. Without this state transformation, system failure is likely to occur. Hence, error recovery is one of the most important aspects of fault tolerance and is one the areas where most work has been done. Different techniques are usually applied for unanticipated damage and anticipated damage.

4) *Fault treatment and continued service:* The first three phases return the system to an error free state. However, an error is a manifestation of a fault, and the removal of the error does not mean the removal of the fault which produced the errors. Unless measures are taken to treat the fault, it may continue to produce errors. This phase provides treatment for the fault itself, and can be divided into two phases. First *fault location* is done, in which the fault is accurately identified. This may not be trivial since the relationship between faults and errors can be complex, and the detection of the error does not necessarily identify or accurately locate the fault. In the second phase of *system repair*, often system reconfiguration or fault repair is done to avoid or remove the fault.

The error recovery phase, unlike the first two phases, is not a passive phase, in that it changes the state of the system. The aim is to eliminate the error from the system state. If the fault is *anticipated* during the design of the system, the damage predicted by the damage assessment phase may be *anticipated damage*[Anderson & Lee 81]. When the fault is *unanticipated*, like a design fault, an accurate prediction of the damage cannot be made and the damage is called *unanticipated*[Anderson & Lee 81]. Anticipated and unanticipated damage usually are handled by different recovery strategies.

The techniques for error recovery can be classified into two categories: backward recovery and forward recovery[Anderson & Lee 81, Randell et. al. 78]. *Backward error recovery* involves backing up of one or more processes in the system to a previous state

which was saved and which is hoped to be error free. The idea is that if a system is restored to a state it occupied prior to the manifestation of a fault, then all errors resulting from that fault will be removed. In contrast, the *forward error recovery* schemes aim to make further use of the state in which the error has been detected. It aims to identify the error and, based on this knowledge, correct the system state containing the error.

Forward recovery measures are usually employed for performing recovery from anticipated damage. Due to the nature of forward recovery, it is dependent on damage assessment and the error identification. Consequently, it is has to be designed specially for a particular system, and is an inappropriate means to recover from unanticipated faults. Due to these it is impossible to implement it as general mechanisms[Anderson & Lee 81].

In contrast, backward recovery is capable of providing recovery from arbitrary and unanticipated faults. It is a general concept applicable to all systems, and can easily be provided as a mechanism. However, due to the state restoration, backward recovery is usually more expensive than forward recovery, and is not applicable to systems which have *unrecoverable objects*[Anderson & Lee 81].

With this background, now we will describe some of the techniques for constructing fault-tolerant software. We will first discuss the different strategies for providing fault tolerance that employ backward error recovery, and then we will discuss the strategies that are based on forward error recovery. Within this broad classification we will separately discuss the techniques for sequential and concurrent systems.

3.2. Strategies Based on Backward Error Recovery

In this section we will discuss the strategies that have been proposed for constructing fault tolerant software, and which employ backward recovery for their error recovery phase. First the recovery block scheme is described, which is applicable to sequential systems. Then the schemes based on the conversation construct are discussed. The conversation construct is an extension of the recovery block scheme to concurrent systems. Finally, other strategies for recovering in concurrent systems, which are not based on the conversation construct, are discussed.

3.2.1. The Recovery Block Scheme

The recovery block scheme[Horning et. al. 74, Randell 75] provides a technique for structuring sequential programs so as to provide means of tolerating faults whose exact location and nature may not be known. It utilizes a backward error recovery method for performing recovery from an error, and so is capable of handling unanticipated faults. Recovery blocks can be considered as providing a means for expressing atomic actions in sequential systems[Randell et. al. 78], and for specifying a final programmed check on the results of an atomic action.

The error recovery scheme depends on the provision of *recovery points.* A recovery point is the program location at which the state of the process is recorded, to which the process may be later reinstated. Different techniques can be used for supporting recovery points. The recovery points are *planned*, that is, the locations at which recovery points are to be established are stated in the program itself.

Error detection in a recovery block is done by an *acceptance test.* Acceptance tests attempt to enforce some standard of behavior lower than absolute correctness, since test-

ing for absolute correctness is impractical. In the recovery block scheme an acceptance test is a function applied to the state of the system after the computation is performed, and tries to determine if the state of the system is acceptable. It is hoped that the acceptance tests would identify states that have major errors. It usually does not try to identify the nature of the error, and the 'failing' of an acceptance test signifies the presence of errors in the state.

If an acceptance test fails, it is assumed that the system component which was responsible for producing that state has some design fault, and so the damage is assumed to be limited to that component. If this assumption turns out to be incorrect, then the system component to which this component belongs is assumed to be having the fault and the damage limited to it. After error recovery is successfully performed, continued service of the component is provided by 'alternates', which act as standby-spares.

The functioning of the recovery block scheme is as follows: a recovery point is established at the start of the recovery block. The first algorithm, called the 'primary', of the recovery block is then executed, followed by the evaluation of the acceptance test of the recovery block. If the acceptance test fails, backward recovery is invoked and the state of the process is reset to the state recorded by the recovery point of the recovery block. Then the next algorithm, called an 'alternate', is executed. Since, the state of the process was reset to the state at the start of the recovery block, everything the primary had done is discarded, and the alternate starts in a hopefully correct state. After the alternate is executed, the acceptance test is again evaluated, and if it fails again, another alternate is executed, again after first restoring the state of the process to its state at the start of the recovery block. If the recovery block runs out of alternates without successfully evaluating the acceptance test, then the recovery block 'fails', and to avoid failure of the system,

recovery and fault tolerance must be attempted at a higher level of abstraction, in some enclosing recovery block.

Because each of the alternates start with the same state at the start of the recovery block, their designs can be independent of each other. The designer of an alternate need not have any knowledge of the design of other alternates, and has no responsibility for coping with any damage that may have been caused by other alternates. For better results it is necessary that the alternates be independently designed, such that the 'fault coverage' increases. If all the alternates have the same faults, recovery block scheme will not be of help. The underlying assumption is that the alternates are independent of each other such that the faults in different designs are not correlated. On subsequent uses of the recovery block, possibly with different input data, the primary is again used.

The recovery block scheme is conceptually quite simple, and the only mechanism needed is the mechanism to create recovery points and reinstate the state of the system automatically, if the acceptance test fails. It can be extended to multilevel systems[Anderson et. al. 78]. An implementation of recovery blocks is described in[Anderson & Kerr 76].

3.2.2. Extensions of Recovery Block in Concurrent Systems

Providing backward recovery in sequential systems is relatively straight forward. Means are needed to record a state and restore a previously recorded state, if needed. However, the situation becomes more complex in a concurrent system, due to the interaction between the different processes of the system. An assumption of backward recovery is that the state to which the system is reverted back, should be error free. In backward recovery, since the cause of the error is not determined, all the computation performed

since the recovery point is considered as suspect, and consequently if a process performs recovery, any other process that may have communicated with the computation that is being discarded, must also recover. For instance, if a process detects an error and rolls back to its nearest recovery point RP, then information which was exchanged between this process and other processes from RP to the error detection point must be revoked. This will cause other processes to roll back. Communication causes dependencies between the processes, which forces a process to recover due to error detections in other processes with which it has interacted.

Let us consider an example. Suppose there are two processes P_1 and P_2, which have the recovery points RP_1 and RP_2. Suppose P_1 receives information from P_2 before RP_1, sent by P_2 after RP_2 has been established. Also suppose P_2 receives information from P_1, after the recovery points RP_1 and RP_2 have been established. If P_1 performs recovery and reverts its state back to the state at RP_1, then the process P_2 will have to recover to RP_2 since it can no longer depend on the communication with P_1, which has been discarded by P_1. Due to the interaction between P_1 and P_2 between the establishments of RP_1 and RP_2, P_1 will have to further perform recovery, and recover to a recovery point earlier than RP_1. So, recovery initiated by P_1, becomes the cause of further recovery by P_1.

The reason of this cyclic dependency for performing recovery is that the recovery points of different processes are not well coordinated with the communication between the processes. Unless the process interactions are properly controlled such that the process interactions and recovery points are well coordinated and cyclic dependencies are avoided, an uncontrolled rollback may result, which is called the *domino effect*[Randell 75].

To restrict communication between processes in such a way that the communication is well coordinated with the setting of recovery points of the processes and the domino effect is avoided, the *conversation* construct was proposed[Randell 75]. A conversation is a two-dimensional enclosure of recoverable activities of interacting processes, and forms a recoverable interacting session[Kim 82]. It is a two dimensional recovery block, which spans two or more processes, and creates a boundary which process interactions may not cross.

The conversation construct works as follows. Each process sets a recovery point when it enters the conversation construct. The set of recovery points at the entry of the conversation together form the *recovery line* of the conversation construct. Once inside the conversation construct, a process may interact only with the activities of the processes that are inside the same conversation, and is not permitted to interact, directly or indirectly, with any process outside the conversation. The part of a process taking part in a conversation forms a recovery block like structure. Each process has its own acceptance test and alternates. When a process evaluates its acceptance test, it is required to wait until the other processes have evaluated their acceptance tests. If *all* the acceptance tests succeed then the processes leave the conversation. However, if *any* of the acceptance tests fail, all the processes inside the conversation must recover to their recovery points at the entry of the conversation. The set of acceptance tests together forms the *test line* of the conversation block.

In essence, the conversation construct is trying to restrict communication between processes such that the computation inside the conversation forms an atomic action. The restrictions imposed ensure that no activity inside a conversation interacts with any activity outside the conversation. Due to this restriction only the processes taking part in

the conversation need to perform recovery.

The basic rules of the conversation scheme can be summarized as follows[Kim 82, Randell 75]:

1) A conversation defines a recovery line as a line which processes do not cross during the rollback.

2) Processes enter the conversation asynchronously.

3) A conversation defines a test line which is an acceptability criterion for the results of the conversation. A test line can be viewed as the acceptance test for the conversation.

4) Process cooperate in error detection, and if an error is detected, regardless of the source of error, all processes must perform recovery.

5) No interaction may take place between activities inside the conversation and the activities outside the conversation block.

6) The conversations may be nested, but only strict nesting is permitted.

Some implementations of the conversation construct have been proposed. In[Kim 82] A proposal has been made for the mechanization of the conversation scheme which is based on monitors. Special monitors are proposed with access restrictions on processes to satisfy the restrictions of the conversation scheme. An implementation of the conversation construct in the language Concurrent Pascal is described in[Shrivastava 79].

3.2.3. Dynamic Techniques

There are two major advantages of the conversation construct. The first advantage is that the recovery line is statically defined. That is, the set of recovery points to which

the processes should roll back, in order to establish an error free state, is simply the set of recovery points set up by the processes when they entered the conversation. As a consequence of the property that the recovery line is statically defined, the amount of recovery needed, in case an error occurs, is fixed and statically determined, and consequently, no domino effect can take place. These properties are obtained by imposing restrictions on the interactions between the processes. These restrictions are reflected as restrictions on the system structure during the design time, and requires the system designer to follow special design restrictions for the construction of fault tolerant software.

Instead of imposing restrictions on process communication, an alternative approach for performing rollback is to keep a record of the process interaction, and based on this record determine which set of recovery points together form the recovery line. With the knowledge of past interaction between processes, if a process is rolled back (due to error detection, or due to the rollback of another process with which it interacted), it can be determined which other processes should roll back. This approach does not need support from the system designer, and recovery can be performed by the underlying system. However, in its simplest form, this approach does not prevent the domino effect from occurring. To prevent the domino effect the state of the processes have to be saved at points other than the recovery points. In this section we will briefly describe some of the proposed methods, which rely on the history of interaction between processes for performing recovery.

Let us first consider those methods which dynamically determine the recovery line, but have no provision for preventing the domino effect. The chase protocols of Merlin and Randell[Merlin & Randell 78] are an example of such a method. The method uses the occurrence graph model to determine the recovery line. In the model the occurrence of an

event is denoted by a 'bar' and the input to the bar indicate which conditions were necessary for the event to occur. The conditions are called 'places'. A restorable condition (a recovery point, for instance) is represented by a restorable place. Simply stating, the detection of an error makes some places and bars invalid. The goal for performing proper recovery is to find a subgraph that contains all the invalidated places and bars, such that there are no outgoing arcs from this subgraph to other subgraphs, and the incoming arcs come directly from restorable places. This subgraph contains the computation to be discarded and the restorable places from which the input arcs to this subgraph start, together form the recovery line.

An occurrence graph represents the actual occurrence of events during execution, and pertinent conditions which actually influence them. Communication between activities of processes will be reflected as input and output arcs from bars. Setting of a recovery point will be represented as a recoverable place. So all the history of process interaction is captured in the occurrence graph, and is used when the recovery is performed. The recovery line is the set of recovery points in processes, such that no process has communicated with any other process after its recovery point (no outgoing arcs from the subgraph). The way recovery line is determined, there is no mechanism to limit roll back, and the domino effect may take place.

The scheme of Wood[Wood 81] is an optimized implementation of Merlin and Randell's scheme. A recovery point RP in a process P is called a direct propagator to a recovery point RP' in the process P', if information flows from P to P', and RP and RP' are active recovery points. The scheme requires that information about direct propagator be kept for each recovery point. If each process is aware of the recovery points to which each of its own recovery points is a direct propagator, then it knows where it should

invoke recovery in the event it has itself to recover. Methods are proposed to efficiently keep this information, and algorithms are given to use the information to determine the recovery line. This scheme too, like the chase protocols, has no means for controlling the amount of roll back, and the domino effect is not prevented.

There are other schemes which dynamically determine the amount of rollback needed, and also prevent domino effect from occurring. The programmer-transparent scheme of Kim is an example[Kim 80, Kim 78] of such a method. However, there is a price for this property. Extra recovery points are set up by the system, besides the one that are defined by the system designer. These extra recovery points called branch-RPs, are established immediately before any interaction (in the scheme the processes interact via monitors), and are responsible for preventing the domino effect. If a process P rolls back, all its interactions are revoked. Any process which had interacted with P only needs to roll back to the branch-RP that was established just before the interaction with P. If no branch-RPs were established other processes will have to roll back to their recovery points, which might revoke other interactions and cause further roll back.

The problem of state restoration has been formally discussed by Russel in[Russel 80]. Propagation of rollback is categorized into two categories, and then conditions are derived under which no domino effect can take place.

3.2.4. The Deadline Mechanism

Specifications of a real-time system often include real-time constraints. Incorrect design or implementation of the system can cause timing faults which can result in system failures. The deadline mechanism[Campbell et. al. 79, Horton 79] was proposed to aid in the design of fault-tolerant real-time systems.

The mechanism is based upon the recovery block mechanism. Two algorithms are provided for each service subject to timing constraints. The "primary" algorithm produces a better quality service than the "alternate". The alternate is a simpler algorithm which produces acceptable results in a known, fixed length of time. The aim is to ensure that either the primary or the alternate completes before the deadline. The acceptance test of the recovery block is replaced by a centralized scheduler and supervisor.

The difference between the deadline time and the time it takes to complete the alternate is called the *Slack time*. The deadline works as follows. First the primary is executed. If it produces acceptable results within the slack time, the program can be exited. However, if the primary module is not completed within the slack time due to some time or design fault, its execution is terminated and the alternate module is invoked (which will complete before the deadline, due to the way slack time is defined). An alternate pessimistic strategy would be to first invoke the alternate module, and store the results it produces in a cache. The primary is then attempted. If the primary fails to complete before the deadline, the results of the alternate are used.

3.3. Strategies Based on Forward Error Recovery

In the schemes based on backward recovery, the state of some component during the error recovery phase is changed to the initial state of that component. A scheme can be regarded as 'forward' if the states of some of the variables involved in recovery are different from their initial states. A forward recovery action is based on the knowledge about the semantics of the component for which recovery is planned, and has to be explicitly programmed by the programmer. Forward recovery also requires accurate damage assessment and the knowledge about the nature of the fault. A framework for supporting forward recovery can be provided by the notion of exception and exception handling. The

framework of exception handling is general enough to handle backward recovery also. In the next section we will briefly discuss the exception handling framework.

3.3.1. Exception Handling

To provide fault tolerance, redundancy must be added into the system. To minimize the increase in complexity due to complexity, the system must be carefully structured and controlled. Some framework is needed within which the four constituent phases of fault tolerance can be implemented, and the abnormal activities (the measures of fault tolerance) of the system separated from the normal activity of the system, and automatically invoked when required. A framework for fault tolerance can be provided by the notion of exception, exception conditions and exception handler. Here we will briefly discuss the exception handling framework described in[Anderson & Lee 81, Campbell & Randell 83].

The abnormal responses of a component are referred to as exceptional responses or *exceptions.* Measures that are provided within the program for dealing with an exception are termed as the *handler* for that exception. In case of an exception, the *exception mechanism* of the system automatically invokes the handler for the exception, by forcing a change in the flow of control of the program. If a service provided by a component is invoked with an invalid set of parameters, the component may return an *interface exception.* Similarly, if a component cannot tolerate a fault it has detected, it may return a *failure exception.* If a component returns an exception, it is said to *signal exception* to the component that requested its service. An interface exception need not always indicate the presence of a fault, while the failure exceptions must be associated with the presence of faults.

If a component either receives an abnormal response from some some component it invoked or detects an error, it *raises an exception*, and invokes fault tolerance measures. Recovery is continued until the component either returns to its normal activities or signals an exception. The flow of control within a component changes to an *exceptional flow of control* as a result of a raised exception, and executes the *exception handler* for the raised exception. Note that an exception that is signaled in a component is handled in the component that invoked it, while an exception that is raised is handled inside the component it is raised.

If fault tolerance measures are successful, a handler may provide a normal control flow return from the component which handled the exception to the component that invoked that component. If the fault tolerance measures are unsuccessful or inadequate, a handler should signal a failure exception. Usually no means are provided to resume the activity that signals the exception, after the exception has been handled. This is called the *termination model*, as opposed to the *resumption model*. It has been argued that the termination model is sufficient for exception handling[Liskov & Snyder 79], though the proposal in[Goodenough 75] has provisions for both.

3.3.2. Exception Handling in Asynchronous Systems

Exception handling in concurrent systems is complicated by the possibility of communication of erroneous information, which requires coordination of processes involved in recovery. The situation is further complicated by the fact that multiple exceptions can simultaneously occur in different processes, and different exceptions in different processes might be caused by the same fault in the system. Generalizing the exception handling framework to concurrent systems requires additional system structure concerning the cooperation and coordination of the individual processes. In this section we describe the

proposal by Campbell and Randell for supporting exception handling in asynchronous systems[Campbell & Randell 83].

Atomic actions are chosen as the basic structure to support fault tolerance. An atomic action is considered as composed of other nested atomic actions called internal atomic actions. An atomic action provides the context for the application of error detection and damage assessment techniques. They propose two principles for structuring fault tolerance within asynchronous systems:

1) The operations provided by a fault tolerant asynchronous system should be implemented by atomic actions.
2) Each fault tolerance measure should be associated with a particular atomic action and should involve all of its processes.

If a component of an atomic action raises an exception, it indicates the detection of an error. The raising of an exception within a atomic action requires the application of abnormal computation and mechanisms to handle the exception. If the recovery measures succeed, the atomic action should produce the results that are normally expected from its activation. If an exception is raised in an internal atomic action, then the fault tolerance measures of that internal action should be applied. However, if the internal action signals an exception, then that exception is raised in the atomic action itself.

The main features of the proposed scheme are described below. If one or several components of the atomic action raise an exception, the fault tolerant measures necessarily involve all the processes of that atomic action. Every component of the atomic action responds to the raised exception by changing to an abnormal activity, and each process whose normal control flow is within one of the components changes to an exceptional control flow, which executes a handler for that exception. This handler either

returns the component to normal activity or signals a further exception.

Each component of an atomic action is restricted to return the same exception. This ensures that the components agree on the abnormal result that should be returned to indicate the failure of the atomic action. If any of the components for an atomic action do not have a handler for the raised exception then all of the components should signal an atomic action failure.

An exception resolution scheme handles the situation when multiple exceptions are raised inside an atomic action. The aim of the resolution scheme is to resolve the multiple exceptions into a single exception for the atomic action, such that all the components can then perform recovery for the same exception. For exception resolution, a exception hierarchy is established. Exceptions are organized in a lattice structure, with *universal exception* as the upper bound of the lattice. In case of multiple exceptions, the resolution scheme returns the exception that is the upper bound of the smallest sub lattice containing all the raised exceptions. Each atomic action has its own exception lattice. The scheme is flexible and can incorporate backward recovery, and use the two recovery schemes in a complementary manner.

3.4. Combined Recovery

It has been suggested that the two recovery techniques should be considered as complementary rather that competitive[Melliar-Smith & Randell 77, Cristian 82, Anderson & Lee 81]. Forward recovery technique uses some part of the current state and tries to modify such that an error free state can result. Since backward recovery is independent of the type of fault and the actual erroneous state, application of forward recovery does not violate the applicability of backward recovery, and backward recovery can success-

fully be applied after forward recovery has been attempted. The class of faults which the two methods aim to handle are also different. Forward recovery is usually applied for anticipated faults, while backward recovery aims to cope with unanticipated faults. So, the two techniques should be considered as complementary, and mechanism for both should be provided.

A framework based on exception handling was proposed by Cristian to incorporate both recovery techniques [Cristian 82], and was later extended to asynchronous systems by Campbell and Randell[Campbell & Randell 83]. The technique is based on failure exceptions and default handlers. If an unanticipated exception occurs (i.e. one which was not anticipated by the programmer) it is called a *failure exception.* By definition of a failure exception, if a failure exception is detected, there may not be a handler to handle the exception. The suggestion is to associate a *default handler*, provided by the system, which is automatically invoked in case of a failure exception.

The default handler must also try to mask the error and try to bring the system to a consistent state, failing which it must signal a failure. Since, a failure exception is an unanticipated error, it can be best handled by backward recovery, and consequently, the default handler should attempt backward recovery. If a recovery block structure is used for backward recovery, a failure exception may be detected because some component used by the primary signaled a failure exception, or because the acceptance test of the recovery block fails. If a failure exception is detected in the primary algorithm, the default handler restores the state from the recovery cache, and executes the alternate algorithm.

The state restoration provides the consistent state, and the aim of the alternate is to mask the failure detected by the primary, by achieving the required post condition in a different way. If the alternate also detects a failure exception, a different alternate is

tried, until the recovery block runs out of alternates. In the case that there are no more alternates left, the recovery cannot be performed in the current recovery block and so the failure of the last alternate results in a signaling a failure exception to the user of this recovery block.

In the framework of exception handling, the two forms of error recovery can be combined. Exception handling for anticipated faults provide the forward recovery, and default exception handling based on automatic backward recovery provide the backward recovery. The two techniques are used in a complimentary manner, since backward recovery is invoked when forward recovery technique cannot be applied, and forward recovery is preferred for anticipated faults.

3.5. Atomic Actions and Fault Tolerance

In this section we will discuss the usefulness of atomic actions for damage confinement and error recovery. Damage confinement is concerned with the structure of activities within a system, and the interaction between the components of the system. Concurrency in a system complicates the problem of damage confinement, due to the ease with which the damage may spread through a system. Atomic actions can be used for structuring the activities of the systems, such that damage confinement is facilitated[Anderson & Lee 81].

By definition of atomic actions, if there is no interaction between the activities of an operation and the activities outside the operation, then the operation is an atomic action. Damage spreads through the system through interaction between activities. So, if the fault occurs inside an atomic action, and is also detected inside the same atomic action, then the damage caused due to the fault is confined to the atomic action. If atomic

actions can be identified in the system, then the damage due to a fault is confined to an atomic action containing all the computation between the first erroneous transition due to the fault and error detection. The problem of damage confinement can then be reduced to finding the 'smallest' atomic action which satisfies this requirement. There are two different strategies to identify such an atomic action.

Dynamic measures start with an initial estimate and then refining it. To implement such techniques, it will be necessary to provide measures to explore, in some fashion, the data of the system. These exploratory measures which attempt to investigate the spread of damage can be viewed as searching for and identifying atomic actions to which the damage is confined, at run time. Atomic actions identified in this manner are dynamically identified atomic actions.

Static damage assessment techniques are often used for *apriori* damage assessment within systems. For these techniques to be successful, the atomicity of the computation to which the damage is confined, is assumed by the system designer, and must be enforced by providing measures and mechanisms. The run time structure that is assumed by the system designer should actually be present during execution. The mechanisms that enforce this structuring by confining the activity of a process, and hence providing constraints on the flow of information within the system enable planned atomic actions to be specified and supported. With planned atomic actions, static assessment of the damage is possible.

Many techniques for supporting fault tolerance have used the property of atomicity. The conversation block has the restriction that there may be no interaction across the boundaries of the conversation. This restriction guarantees the atomicity of the computation performed inside the conversation. The recovery block in sequential systems, and the

conversation block in concurrent systems, both form atomic actions with recovery primitives. The dynamic techniques for backward recovery aim to identify that part of the computation which had no interaction with the rest of the computation. This approach essentially tries to identify atomic actions dynamically[Anderson & Lee 81, Best & Randell 81]. For forward recovery in asynchronous systems the approach proposed by Campbell and Randell[Campbell & Randell 83] utilizes the framework of planned atomic actions.

So, atomic actions are useful for damage confinement, and performing recovery. Many techniques utilize the atomicity criteria for providing fault tolerance. In the next chapter we will show that atomic actions are actually necessary for performing backward or forward recovery.

CHAPTER 4

ATOMICITY OF ACTIONS AND RECOVERABILITY

In this chapter we discuss the relationship between atomicity and recoverability of actions. A model of computation is developed in which actions are defined; this definition is applicable both to sequential and to concurrent systems. For this model, atomic actions are defined as actions which satisfy certain conditions. Restrictions on actions that make a general action an atomic action are specified. Recoverability of actions is defined using this model and then the relationship between recoverability and atomicity of actions is shown.

4.1. System Model

In a software system, the activities of the system are generated by by system components referred to as processes. A process can be thought of as the sequence of primitive operations generated when a program or set of programs is executed[Anderson & Lee 81]. A primitive operation is an operation that is indivisibly executed by the hardware. We consider a system consisting of a set of communicating processes. No assumption is made about the relative speeds of processes. The processes communicate with each other, and a process may affect control of another process. No form of interprocess communication is assumed. Processes may communicate through shared memory or by message passing. We assume an interleaved model of computation; the execution of the primitive operations of different processes may be interleaved. An action is generated when sequences of primitive operations are executed by some processes. For the purposes of fault detection and recovery, an action is considered as the basic unit of computation.

The state of a concurrent system at any instant is specified by a pair (S,C), where S is the memory state of the system, and C is the control vector. The *control vector* C of the system is the set of control points, one for each process in the system. The control point for a process at any instant is specified by the contents of its program counter, that is, the location of the statement to be executed next by the process. The *memory state* of the system is specified by the set of values of all the variables in all the processes of the system at the time instance under consideration. This definition of memory state is similar to the one proposed in[Hoare & Lauer 74]. The value of any variable can be retrieved from the memory state by using appropriate retrieve functions. The value of a variable x in a state S will be denoted by $x(S)$. A similar definition for state for a concurrent system has been used in [Lamport 83, Owicki & Lamport 82]. Wherever clear and unambiguous, we will use the term state to refer to the memory state of the system.

Definition 4.1 : A state (S,C) is defined to be a *viable state* as follows:

(a) The initial state of the system is a viable state.

(b) Starting from the initial state, there exists some *normal* execution of processes which will bring the system to the state (S,C).

Viable states are the states which the system may assume during a normal execution. A normal execution is characterized by an execution that has no erroneous transitions [Anderson Lee 1981 It is assumed that the system is such that starting in a non-viable initial state some normal execution of the system would lead to a non-viable system state. For the purposes of recovery, we note that a state of the system after a succession of states containing errors may be a valid system state [Anderson Lee 1981 This occurs in the case an error is detected and a successful recovery from the error is per-

formed. Such a state will also be a viable state, as some error free computation of the system could take the system to this state.

4.2. Actions

We regard an action as having *statically defined* boundaries. The primitive operations of different processes that are enclosed inside these boundaries comprise the body of the action. A computation of the action is an execution of the primitive operations in its body. For example, in sequential systems actions may correspond to a statement, a block, a procedure, or a control structure. This view of an action leaves the definition of the boundaries of an action to the user. However, the boundaries of an action may not be chosen arbitrarily. The restrictions on the boundaries of actions are described later. We define an action for a concurrent system as follows.

Definition 4.2 : An action A is a 3-tuple.

$$A = (I, F, P)$$

where,

I is the initial boundary of A
F is the final boundary of A
P is the set of processes taking part in A.

The initial boundary I, is the set of *initial points*, one for each process p in P. The initial point of a process p for the action A specifies the start of its computation contributing to the computation of the action A. Similarly, F is a set of *final points*, one for each process in P. The final point of a process p specifies the end of the computation contributing to the computation of the action A. The initial point of p will be represented as $p(I)$ and the final point as $p(F)$.

We define the *state variable set* of an action A, as the set of all the variables in the domain of the computation of A. The initial state $S_i(A)$ of an action A is the set of

values of the variables in the state variable set of A just after each of the processes has crossed its initial point. Similarly, the final state $S_f(A)$ is the state of the variables in the state variable set of A just before the processes cross their final point. If the action A is *well-defined*, the initial and final states of A will be uniquely defined for each computation of A. In our model we assume that the actions are well-defined.

During the computation of an action the processes taking part in the action need not cross their respective initial and final points synchronously. This gives rise to the possibility that the variables in the state variable set of A have different values at initial points for different processes. For well-defined actions the boundaries are chosen in such a way that the initial and final state of the action are defined, and the states of the variables in the state variable set of A have the same values at the initial (final) point of each process taking part in the action. The initial (final) state of an action can then be regarded as the state of the variables in the state variable set of A when any one of the processes taking part in the action is at its initial (final) point. Moreover, the initial (final) boundary should be such that the control points of the processes taking part in the action can simultaneously be at their respective initial (final) points. However, the definition imposes no restriction on how the computation of an action may interact with the activities outside the action.

An action A can then be regarded as an operation that, when executed, transforms the state from $S_i(A)$ to $S_f(A)$. There may be intermediate states in transition from $S_i(A)$ to $S_f(A)$. We can, therefore, represent the computation of an action A as a transformation sequence of the form

$$A = S_i(A),\ s_1(A),\ s_2(A),\ \ldots,\ s_n(A),\ S_f(A)$$

The states $s_i(A), 1 \leq i \leq n$ are *intermediate states*. In the case when there is no intermediate state (n=0), the action is a primitive operation which is executed indivisibly by the hardware. For the rest of the chapter we will refer to the transformation sequence representing the computation of an action as the transformation sequence of that action. Due to the non-determinism introduced because of concurrency, many transformation sequences of an action are possible, even for a given initial state.

We assume that computation of an action modifies some variables in its state variable set, and the final state of an action is not same as its initial state. This assumption eliminates the trivial case where the initial state is same as the final state. However, the assumption also disallows an action consisting of serially reusable processes, which only have effect on the outside world. As we are interested in modeling the system and not its effect on the outside world, such processes are not of interest to us.

The states in a transformation sequence of an action are *partial states* of the system, since they define the values of a subset of the variables in the system. Since we often need to consider the state of the system, we need to define the state of the system for these partial states. For this purpose, we consider a partial state as a system state in which the variables that do not contribute to the partial state are labeled as *don't care variables*. Two partial states are *disjoint* if every variable is labeled don't care in at least one of the two states. We have the following definition of compatibility of two partial states.

Definition 4.3 : Two partial states S and S' are said to be *compatible* if every variable in S (S') either has the same value as in S' (S), or is labeled don't care in either or both of S and S'.

Because of the don't care variables there might be many partial states that are compatible to a partial state. If a partial state S' is obtained from a partial state S by assigning a value to a don't care variable in S, then S and S' will be compatible states.

The notion of compatible partial states is a generalization of the notion of equivalent states. If no variable in the partial states is labeled don't care then two partial states are compatible if and only if they are equivalent. We can represent this as a binary relation on states of the system. This relation is symmetric and reflexive. However, unlike the equivalence relation, in general the compatibility relation is not transitive. That is, if we have the partial states S_1,S_2,S_3, then S_1 is compatible with S_2 and S_2 is compatible with S_3, does not imply that S_1 is compatible with S_3.

4.2.1. Nesting

An action may have other actions nested inside it. If an action A' is *nested* within an action A, we write this as $A' \subseteq A$. If A' is an action outside A, we write this as $A' \not\subseteq A$. Nesting allows an action to be composed of many, possibly concurrent, actions. A nested action must be correctly nested, that is, no boundary of the sub action may cross any boundary of the parent action. An action $A'=(I',F',P')$ is said to be nested within another action $A=(I,F,P)$, if it satisfies the following:

(1) $P' \subseteq P$

(2) For any process $Q \in P'$, the initial point of Q for A' does not occur before the initial point of Q for A.

(3) For any process $Q \in P'$, the final point of Q for A' does not occur after the final point of Q for A.

The semantics of nested actions require that an action finishes only *after all* its sub actions have completed their computation. Nested actions aid in decomposing activities in a modular fashion, and are important in designing concurrent activities.

Let $A' \subseteq A$, and let the transformation sequence for A' be

$$A' = S_i(A'), s_1(A'), s_2(A'), \ldots, s_n(A'), S_f(A')$$

Since A' is nested within A, this transformation sequence must also take place in the transformation sequence of A. We can now specify the condition of correct nesting in terms of the state transformation sequences. The action A' is correctly nested within the action A if

(1) For every state $s_i(A'), 1 \leq i \leq n$ in the state transformation sequence of A', there exists a state $s_j(A)$ in the transformation sequence of A, such that the two states are compatible.

2) There exists a state $s_p(A)$ compatible with $S_i(A')$, and another state $s_q(A)$ compatible with $S_f(A')$, such that $p < q$.

The first condition states that every partial state in the transformation sequence of A' exists as a partial state in the transformation sequence of the action A. The second condition state that the initial state of A' appears as a partial state in the transformation sequence of A before the final state of A' does. These conditions ensure that the computation of the action A can finish only after A' has finished its computation.

4.3. Atomic Actions

An atomic action is an operation that appears primitive and indivisible to any activity outside the atomic action. In defining an action, no restriction was imposed on interactions between different actions. For actions to be atomic, interactions between different actions have to be restricted. To define atomic actions we need the following definitions. In these definitions, the initial state of an action A is represented by $s_0(A)$ and the final state as $s_{n+1}(A)$. The variables that are in the state variable sets of two actions A and A' are called common variables.

Definition 4.4 : An action A' *uses* a state $s_j(A), 0 \leq j \leq n+1$ of an action A, if $S_i(A')$ and $s_j(A)$ are not disjoint, and $s_j(A)$ is compatible with $S_i(A')$ and is compatible with $S_f(A')$.

Definition 4.5 : An action A' *modifies* a state $s_j(A), 0 \leq j \leq n+1$ of an action A, if $S_i(A')$ and $s_j(A)$ are not disjoint, and $s_j(A)$ is compatible with $S_i(A')$ and there exists $s_k(A')$ such that $s_k(A')$ is not compatible with $s_j(A)$.

If the actions compute sequentially or compute at different times, atomicity of actions is trivially satisfied. Hence, for the purposes of defining atomic actions we are interested in actions that compute concurrently and can interact with each other in a fashion that violates atomicity. We assume that if there are many states of an action compatible with another state, then the final state is chosen over the initial state and the initial state over the intermediate states.

We consider an action to be computing if the control of at least one of the processes taking part in the action is between its initial and final point. Two actions A and A' are said to be concurrent, if there is some time instance when both the actions are computing. We can specify this notion of concurrent actions using our system model. For a process

p, let the value $p(C)$ reflects its control point in the system control vector C.

Definition 4.6: Two actions $A=(I,F,P)$ and $A'=(I',F',P')$ are *concurrent* if, $A' \not\subseteq A$, and there exists some system control vector C such that for some process $p \in P$, and for some process $p' \in P'$, $p(C)$ is between $p(I)$ and $p(F)$, and $p'(C)$ is between $p'(I')$ and $p'(F')$.

With these definitions we now define an atomic action. Let A be an action represented by the transition sequence

$$A = S_i(A),\ s_1(A),\ s_2(A),\ldots,\ s_n(A),\ S_f(A)\ .$$

Definition 4.6 : An action A is an *atomic action* if there exists *no* action A', such that A' and A are concurrent, and A' modifies $S_i(A)$, or uses some intermediate state $s_j(A)$, $1 \leq j \leq n$.

The requirement that A' not use $s_1(A),\ldots,s_n(A)$ preserves the indivisibility of A. The only states seen outside A are the initial and final states of A, which agrees with the view that an atomic action should appear to transform the state without any intermediate states. The lost update problem [Gray 78, Bernstein & Goodman 81] is avoided because A' is not allowed to modify $S_i(A)$. If A' can modify some values in $S_i(A)$, and if A also modifies the same values of $S_i(A)$, then the final state resulting after the execution of both A and A', will not reflect the modification of one of the actions.

4.3.1. Nesting

The three conditions for correct nesting of general actions will hold for atomic actions also. However, due to the atomicity of sub actions, the conditions on transformation sequences of atomic actions are different from the conditions for general actions. Let

A and A' be atomic actions, such that $A' \subseteq A$. By definition of atomicity no intermediate state of A' may be used by A. For the initial and final state of A', we have the following conditions.

(1) $S_i(A')$ is compatible with $s_k(A)$, for some k.

(2) $S_f(A')$ is compatible with $s_l(A)$, for some $l > k$.

These conditions imply that the initial and final states of A' are reflected in the transformation sequence of A. Atomicity of A' prevents the intermediate states from being used by A, so the condition that each of the intermediate states of A' also be reflected in the transformation sequence of A is not required for atomic actions.

In the above situation, the atomic action A' modifies or uses some intermeditate state in the state transformation sequence of the action A. However, this is permissible by the definition of atomic actions, because the restrictions on the states in the transformation sequence of A are only for actions A' such that $A' \not\subseteq A$.

4.3.2. Strict Sequencing Property

Atomic actions should appear to act in strict sequence, even if they execute concurrently. This is the strict sequence property of atomic actions[Lomet 77] and is often referred to as the serializability criteria[Eswaran et. al. 76, Liskov & Scheifler 83]. Let $A_1, \ldots, A_n$ be atomic actions which execute concurrently such that $A_i \not\subseteq A_j, 1 \leq i,j \leq n$. According to the strict sequencing property, when all the actions have terminated, the resulting state should be equivalent to executing the actions serially in some order. The serial order to which the execution is equivalent will be called an Equivalent Serial Order (ESO). This means that the state of the system obtained by executing the actions is compatible to the state obtained by executing the actions sequentially in the order specified

by the ESO. There might be more that one ESOs for the execution of actions.

The execution of the actions can be checked for serializability using its corresponding *precedence graph*[Ullman 80]. We construct the graph precedence graph G_P as follows. The nodes correspond to the actions. The arcs are determined by the following rules. The variables that are in the state variable set of both A_j and A_k will be referred to as common variables.

(1) If A_j uses $S_i(A_k)$, and A_k modifies some common variables, draw an arc from A_j to A_k.

(2) If A_j uses $S_f(A_k)$, draw an arc from A_k to A_j.

(3) If A_j modifies $S_f(A_k)$, draw an arc from A_k to A_j.

If the state variable sets of two actions are disjoint then there is no arc between the actions in the precedence graph. Moreover, if two actions use the initial state of each other, the situation is similar to the read-read dependency situation in databases[Bernstein & Goodman 81], and no arc is created. By definition, if any action A_j uses the initial state, or uses or modifies the final state of another action A_k, the state variable sets of A_j and A_k are not disjoint.

Lemma 4.1: In the graph G_P, if an arc is created by rule (1) from an action A_j to A_k, an arc will also be created from A_j to A_k by rule (3).

Proof: A_j uses $S_i(A_k)$, therefore, $S_i(A_k)$ is compatible to $S_f(A_j)$. Since, A_k modifies some of the common variables, $S_f(A_k)$ is not compatible with $S_f(A_j)$. This implies that A_k modifies $S_f(A_j)$, and this will cause an arc from A_j to A_k by rule (3).

□

From the precedence graph, G_P, we construct the reduced precedence graph, G_R, by deleting the arcs formed due to rule (1). In a precedence graph, we say a node n_1 precedes another node n_2, if there is an arc from n_1 to n_2. We say that two graphs are equivalent, if they have the same set of nodes, and if a node n_1 precedes another node n_2 in one graph, then n_1 also precedes n_2 in the other graph.

Lemma 4.2: The reduced precedence graph G_R and the precedence graph G_A are equivalent.

Proof: Follows from Lemma 1.

□

Definition 4.7: For a sequence of partial states $S_1, S_2, \ldots, S_p$ we define the *composite state* $C(S_1, \ldots, S_p)$ as the state in which the value of a variable x is $x(S_j)$, $1 \leq j \leq p$, such that for all $k > j$, $x(S_k)$ is labeled don't care. If x is labeled don't care in all the S_j, $1 \leq j \leq p$, then it is labeled don't care in the composite state also.

If an action A_1 modifies a state S and another action A_2 modifies or uses the final state of A_1, then A_2 is actually modifying or using the composite state $C(S, S_f(A_1))$. That is, for a variable x which is labeled don't care in $S_f(A_1)$, but has a value assigned in S, the value of x used by A_2 is $x(S)$. However, if x has a value assigned in both the states then the value in $S_f(A_1)$ is used, since it is the "most current" value.

To show the strict sequence property we will draw a precedence graph for the atomic actions. Let $A_1, A_2, \ldots, A_n$ be a set of actions which execute concurrently. We assume the existence of an action A_0 which precedes every action and assigns value to all the variables in the state variable set of the actions. This will ensure that if a node A_i in the precedence graph does not have an arc from any other node, then there will be an arc

from the node A_0 to A_i. This ensures that the precedence graph is fully connected and there is an incoming edge for each node, except A_0. We construct the reduced precedence graph G_R for the atomic actions as described earlier in the section.

Lemma 4.3: Let $A_1;A_2;...;A_p$ be a path in the graph G_R, and let $S=C(S_f(A_1),...,S_f(A_j))$. Then S is not compatible to the states $S_i(A_k),1\leq k\leq j$.

Proof: For an action, the initial and final states are not compatible. The actions $A_k,...,A_p$ can be considered as sub actions of an enclosing action, with the initial state as $S_i(A_k)$ and the final state as S. It follows that $S_i(A_k)$ and S are not compatible.

□

Theorem 4.1: The reduced dependence graph G_R is acyclic.

Proof: The proof is by contradiction. Let there be a cycle $A_1;A_2;\cdots;A_p;A_1$. By the second occurrence of A_1 in the cycle, $S_i(A_1)$ is compatible to the composite state $S=C(S_f(A_1),S_f(A_2),...,S_f(A_p))$. By Lemma 4.3, the composite state S is not compatible to $S_i(A_1)$. This is a contradiction. Hence the assumed cycle cannot exist.

□

Hence, the dependence graph of the execution of atomic actions is acyclic. This ensures that there is always a serial order of execution of actions, to which an execution is equivalent. Hence, actions which satisfy the definition of atomicity, also satisfy the strict sequence requirement.

4.4. Recoverability of Actions

The system model of actions and concurrent processes can be applied to fault tolerant systems, to investigate the role of atomic actions in recovery schemes. We

assume that an action is the unit within which a user will specify fault-tolerance. Since our definition of an action is very general, this assumption does not impose a restriction on choosing appropriate units for recovery. Nested actions can support nested recovery schemes. We assume that if the recovery measure for the action is successful, the final state of the action that is produced is one which would result if the action had performed a correct execution.

We require that recovery be *planned recovery*, that is, the recovery methods to be applied are decided at design time, and the recovery methods have a predictable effect on the system state. That is, in case of an error during the computation of the system, the recovery methods to be applied can be determined statically, and, for a given initial state, the state of the system after the recovery is performed, can be predicted *apriori*. The model of planned recovery disallows a recovery technique that is non deterministic, or a recovery technique that invokes an oracle to perform recovery.

We represent a *successful recovery sequence* of the action A as:

$$S_i(A) \dashrightarrow s_j(A) \dashrightarrow recovery \dashrightarrow S_f(A)$$

This sequence represents a computation of A in which a successful recovery is initiated at $s_j(A)$. We want that every successful recovery sequence should result in a viable system state. In that case, the function of the fault tolerant technique is fulfilled, by taking the system to a state it could have reached by a normal computation, and from where normal processing can again be resumed. This notion of recovery sequence is abstract and permits a general form of error detection at any intermediate state in the state transformation sequence. For the purposes of our arguments, we assume that after the recovery is performed there exists a system state (S,C) such that S and $S_f(A)$ are compatible.

4.4.1. Backward Recoverability

Our model of backward recovery is based on the recovery block model[Randell 75]. To perform backward recovery for an action, we assume that each process taking part in the action saves its state at its initial point for the action. By doing this, the initial state of the action is saved. In case an error is detected, each process restores the saved state, changes its control point to its initial point, and then perform an alternate computation. As in the recovery block scheme, the alternate computation is specified statically, and has a predictable effect on the state of the system. For a successful backward recovery in an action the initial state of the action is assumed to be error-free.

A successful recovery sequence of this method for an action A is of the form

$$S_i(A) \dashrightarrow s_j(A)\, S_i(A) \dashrightarrow S_f(A)$$

This sequence has two phases. The recovery measure first performs state restoration and restores the state of the action to its initial state. In the second phase, an alternate computation is performed which produces a valid final state of the action, for the given initial state of the action. This sequence represents a successful completion of backward recovery for the action A. Note that if recovery is invoked, it implies that there must be some intermediate states in the state transformation sequence which are erroneous. For this model of backward recovery we define the backward recoverability of an action as follows.

Definition 4.8 : An action A is *backward recoverable* if a successful completion of backward recovery in action A *always* results in a viable system state.

Before we relate backward recoverability of an action with the atomicity of the action, we would like to show that the system state must be viable after the state restora-

tion, for A to be recoverable. For this we need to show that if the system state after state restoration is not viable, there always is some recovery sequence of A, which would take the system to a state that is not viable. This implies that a completion of backward recovery of A does not always lead to a viable system state.

Lemma 4.4 : The condition that the system state is viable after state restoration is necessary to ensure that an action A is backward recoverable.

Proof : Let the system state after the state restoration in A be (S,C). S is compatible to $S_i(A)$. Assume that after the state restoration is performed, all the processes not taking part in A do not perform any computation until the recovery of A is complete. Let the system state after a successful completion of backward recovery in A be (S',C'). S' is compatible to $S_f(A)$. Any execution of the system reaching (S',C') must execute the action A in the initial state $S_i(A)$. Furthermore, the only way to advance the system state from (S,C) to (S',C') is by performing an error free computation of A (which in the case of successful backward recovery is the computation of the alternate algorithms). Consequently, if (S,C) is not a viable state, (S',C') is not viable. By assumption about the normal computation, if the system state after state restoration is not viable, there exists an execution in which the completion of backward recovery does not lead to a viable system state.

□

As a consequence of this lemma, it is sufficient to show that a non-viable system state can occur after state restoration within an action to conclude that the action is not backward recoverable. In a state (S,C), we will use the notation $S(A)$ to represent the state of the variables in the state variable set of the action A. The definition of backward

recoverability and atomic actions can now be compared.

Theorem 4.2 : If an action A is backward recoverable, then A is an atomic action.

Proof : We show this by proving the contrapositive of the theorem, that is, if A is not atomic, it is not recoverable. Let (S,C) represent the state of the system after state restoration is performed. There are two cases.

Case 1: Some action A', $A' \not\subseteq A$, modifies the initial $S_i(A)$. Although $S(A)=S_i(A)$ the value of C implies that A' has been executed (so, the effects of A' are lost). Therefore, a normal computation of the system will not produce (S,C).

Case 2: Some action A', $A' \not\subseteq A$, uses $s_j(A)$, and $s_j(A)$ is erroneous. Then no normal execution will generate the state $s_j(A)$. Consequently, a normal computation will not produce (S,C).

So, if A is not an atomic action, the state (S,C) may not be a viable state. By lemma 4.1, this implies that A is not backward recoverable.

□

Here we have shown that atomicity of the action is needed for state restoration. Since state restoration is the first step after error detection, atomicity is essential for any fault tolerant method employing backward error recovery techniques. It should be pointed out that if A' also performs backward recovery, then the system state *after both* actions have performed state restoration, could be a viable state. In this case, backward recoverability and state restoration should be considered as being performed in an enclosing action A'', composed of A and A'. In case 2 of the proof, $s_j(A)$ is assumed to be erroneous. If $s_j(A)$ is not erroneous, the argument will not be true. However, since an

error was detected, there must be at least one intermediate state that is erroneous. Consequently, there is always a case (when that erroneous state is used) where lack of atomicity will lead to a non viable state. By our definition of recoverability, this implies that the action is not recoverable.

4.4.2. Forward Recoverability

In contrast with backward recovery, the forward recovery methods do not save the previous states the system. Forward recovery techniques manipulate some portion of the state at which the error is detected to produce a new state, in the hope that the new state will be error free. The activities that are performed during forward recovery are dependent on the type of error and the operation in which the error occurred.

Consequently, there are no general rules to predict the nature of the state manipulation which might be done by a forward recovery scheme. For example, forward recovery can also "undo" a command, though this "undoing" has to be done by performing operations on the erroneous state, and no previously stored state can be used. For example, for an erroneous insert command, the forward recovery might execute a delete command to "undo" the previous insert command.

In performing forward recovery on an action A, if some error is detected at state $s_j(A)$ in the transformation sequence of A, then a different transition sequence will begin from $s_j(A)$. The successful recovery sequence of an action A for forward recovery is of the form

$$S_i(A),\ s_1(A),\ldots,\ s_j(A),\ s_{j1}(A),\ s_{j2}(A),\ldots,\ s_{jm}(A),\ S_f(A)\ .$$

The state transition sequence $s_{j1}(A),\ s_{j2}(A),\ \ldots,\ S_{jm}(A), S_f(A)$ is the result of the activities during the forward recovery, and is called the *exception sequence*. The state of

the action after a successful recovery is performed is an acceptable final state for that action. For this model of forward recovery we define the forward recoverability of an action as follows.

Definition 4.9 : An action A is *forward recoverable* if a successful completion of forward recovery in the action A, *always* results in a viable system state.

Theorem 4.3 : If the action A is forward recoverable, then A is an atomic action.

Proof : We show this by proving the contrapositive of the theorem. Let (S,C) represent the system state after the recovery on A is completed. There are two cases.

Case 1: Some action A', $A' \not\subseteq A$, modifies $S_i(A)$. Let the exception sequence modify the variables modified by A' in a different manner than the normal sequence of A. Consequently, a normal computation of the system will not produce (S,C).

Case 2: Some action A', $A' \not\subseteq A$, uses $s_j(A)$, and $s_j(A)$ is erroneous (later 'corrected' by the exception sequence). No normal execution will generate the state $s_j(A)$. Consequently, a normal computation will not produce (S,C).

So, if A is not an atomic action, the state (S,C) may not be a viable state. Thus, A is not forward recoverable.

□

Since forward recovery methods are system and error specific, it is difficult to reason about forward recoverability and the proof of the above theorem depends on assumptions about the exception sequence. These assumptions might not always be true, and consequently, there might be cases of forward recovery where atomicity of the action might be an overly strict requirement. However, if the action is not atomic, there would

be cases in which a successful recovery for the action will not lead to viable system state. Since our definition of recoverability requires that every successful recovery for an action leads to a viable system state, to be able to support a general forward recovery for an action, atomic actions are necessary. Atomic actions provide the structural framework within which any forward recovery can be programmed, without worrying about the effect of the recovery measure on the actions outside the action in which the recovery is being performed.

4.4.3. Combined Recoverability

It has been suggested that the two techniques for providing fault tolerance should be used in a complimentary fashion to take the benefits of both[Randell et. al. 78]. Different schemes to combine the two forms have been proposed [Cristian 82, Campbell & Randell 83]. The details of the schemes for combining the two forms of recovery need not concern us here. However, if both forms of recovery are to be performed for an action, then the action should be both forward and backward recoverable. Recoverability of one type does not necessarily imply the other. For the combined recoverability of actions we have the following definition.

Definition 4.10a: An action A is recoverable if it is forward recoverable *and* backward recoverable.

The above definition is motivated by the goal of supporting both forward and backward recovery in a complimentary manner. However, there is another view which can be taken for recoverability of actions. From the point of view of an activity which is outside the action in which recovery is being performed, the type of recovery measure being employed is not of importance. What is important is whether the action will be success-

ful in performing recovery or not. With this view point, a recoverable action can be defined as follows.

Definition 4.10b: An action A is recoverable if it is backward recoverable *or* forward recoverable.

Irrespective of the definition of recoverability chosen, we have the following theorem, which trivially follows from the previous theorems.

Theorem 4.4: If an action A is recoverable, then A is an atomic action.

4.5. Discussion

In this chapter we have proposed a model for understanding atomicity and recoverability of actions, in a possibly concurrently environment. In the model, an action is defined as a static construct with statically defined boundaries, and is characterized by a state transformation sequence. The boundaries of an action are not allowed to be chosen arbitrarily, but are required to be chosen such that the initial and final state of the action are always defined. This is a preliminary model aimed to explore the relationship between atomicity and recoverability. Restrictions have to be loosened, and the model made more formal for the results to have wide applicability.

In this model, an atomic action is defined as an action, with restrictions on how other actions may access the states in its state transformation sequence. The definition of atomic actions is shown to have the strict sequence property. The model is used to define forward and backward recoverability of an action. Only planned recovery is considered. It is then shown that if an action is backward or forward recoverable, it is necessarily an atomic action. Consequently, the problem of providing recovery can be divided into two

disjoint problems, the problem of providing atomic actions and the problem of providing recovery. Thus, we propose that the continued study of recovery techniques should be discussed within the framework of atomic actions, and the design of recovery techniques should be independent from concerns of atomicity.

CHAPTER 5

ATOMIC ACTIONS FOR FAULT TOLERANCE USING CSP

Several practical techniques for the construction of fault-tolerant software have evolved[Randell et. al. 78]. The aim of these techniques is to ensure that the system provides the intended service despite possible software or hardware faults. The techniques depend upon two complementary approaches to fault-tolerance known as *forward error recovery* and *backward error recovery* and it has been suggested that both be used to provide more reliable software[Anderson & Lee 81, Campbell & Randell 83, Cristian 82].

Few implementations permit both approaches to be combined within a particular application. Fewer techniques are available for the construction of fault-tolerant software for systems involving concurrent processes and multiple processors. In this chapter, we propose a scheme to support backward and forward error recovery in a system of Communicating Sequential Processes (CSP)[Hoare 78] based on the framework of atomic actions. The atomic action is used as the basic unit for providing fault tolerance. The atomic action is called a FT-Action, and both forward and backward error recovery are performed in the context of a FT-Action. An implementation for the FT-Action is proposed, which employs a distributed control, uses CSP primitives, and supports local compile and run-time checking of the forward and backward error recovery schemes. The results of this chapter have been reported in[Jalote & Campbell 84].

5.1. Communicating Sequential Processes

CSP was proposed by Hoare as the basis for a concurrent programming language. Dijkstra's guarded commands[Dijkstra 75] are used as sequential control structures, and

as the sole means of introducing and controlling nondeterminism. A parallel command specifies concurrent execution of its constituent sequential commands (processes). All the processes start simultaneously, and the parallel command terminates successfully only if and when they all have successfully terminated. Simple forms of input and output commands are introduced which are used for communication between the concurrent processes. CSP processes may only communicate with each other using the input and output message commands. Messages are passed through named synchronous *static channels.* An *output command* is of the form:

destination ! expression

where *destination* is the process name and *expression* is a simple or structured value. An *input command* has the form:

source ? target

where *source* is a process name and *target* is a simple or structured variable.

Communication occurs between two processes of a parallel command whenever (1) an input command in one process specifies as its source the process name of the other process; (2) an output command in the other process specifies as its source the process name of the first process; and (3) the target variable of the input command matches the value denoted by the expression of the output command. On these conditions, the input and output commands are said to *correspond.* An input command fails if its source is terminated. An output command fails if its destination is terminated or if its expression is undefined.

Commands which correspond are executed simultaneously, and their combined effect is to assign the value of the expression of the output command to the target variable of the input command. There is *no* automatic buffering, and an input or output command is delayed until the other process is ready with the corresponding output or input command. A communicating process may wait forever if another process does not match its command. This inherent limitation of a synchronous message passing scheme makes detection of a so called "deserter"[Kim 82] or dead process difficult. After the communication, both processes proceed independently and concurrently.

Dijkstra's Guarded Commands[Dijkstra 75] are used in CSP in the form:

$$G \rightarrow C$$

where G is a guard consisting of a list of Boolean expressions followed by an optional input command list, and C is a command list. Output commands may not appear in the guards. If an input command appears in a guard, it is called an input guard. A guarded command may be executed if and when the execution of its guard does not fail. First, the guard is evaluated by determining the value of its Boolean expressions. If any expression is false, the guard fails; but a guard that evaluates to true has no effect. An input guard may be evaluated only if and when there is a corresponding appropriate output.

The alternative command may be executed by a sequential process. It has the form:

$$[\, G_1 \rightarrow C_1 \,\square\, G_2 \rightarrow C_2 \,\square \ldots \square\, G_n \rightarrow C_n \,]$$

and selects the execution of exactly one of the constituent guarded commands. If all the guards fail, the alternative command fails. Otherwise a command is selected nondeterministically from those commands with successful guards. In the case when more

than one command list can be selected, the choice is nondeterministic. If several of the input guards in an alternate command correspond with output commands elsewhere, only one is selected and the others have no effect.

The notation $(i{:}1..n)G \rightarrow C$ represents the alternative command

$$[\ G_1 \rightarrow C_1 \ \square\ G_2 \rightarrow C_2 \ \square\ \ldots\ \square\ G_n \rightarrow C_n\ ]$$

where each $G_j \rightarrow C_j$ is formed from $G \rightarrow C$ by replacing every occurrence of the bound variable i by the numeral j.

A repetitive command specifies as many iterations as possible of its constituent alternative command. It has the form:

* [alternative command]

When all the guards fail, the repetitive command terminates. Otherwise, the alternative command is executed once and the whole repetitive command is executed again. A repetitive command may have input guards. If all the sources named by the input guards have terminated, then the repetitive command also terminates.

It is possible to program coroutines using the CSP notation, and consequently subroutines can also be programmed. The provision of output commands within the guards has been advocated in[Bernstein 80, Silberschatz 79]. We will assume a version of CSP with both this facility and a basic exception mechanism for a single process.

5.2. Design of the FT-Action

The FT-Action should be designed so that the atomicity of the FT-Action is guaranteed. The atomicity guarantee permits the programming of recovery for the con-

struct. The scheme should be able to support both the programming of backward error recovery and forward error recovery. For backward recovery we employ a conversation-like scheme, which can be easily implemented in a planned atomic action framework. For forward recovery we use the scheme proposed by Campbell and Randell which is based on planned atomic actions.

We define a *Fault Tolerant Atomic Action (FT-Action)* as a distributed control structure that a group of processes may join or leave together in synchrony. Inside an FT-Action the processes may communicate with one another, but not with processes not in the control structure. The FT-Action will be used as a framework within which error recovery can be provided. It has the following properties.

Atomicity: The communications of processes in the control structure must be isolated from other processes to guarantee atomicity and prevent *information smuggling*[Kim 82]. Hence, in a FT-Action, no communication may take place across the boundaries of the FT-Action, and the corresponding command for an input or output command inside the FT-Action must also be inside that FT-Action.

A recovery line for backward error recovery: In the event of an error, the processes may be rolled back to the recovery points that were established at the recovery line. The FT-Action provides a recovery line which is defined by the synchronized entry of all the participating processes.

A test line for the processes: The test line is a set of diagnostic tests, one for each process, which is used to determine whether any errors have occurred. In the FT-Action the exit statements (see next section) in the constituent processes together form a test line. The processes synchronize at the test line.

Recovery measures: The FT-Action should have provision and primitives for performing recovery. If any process detects errors inside the FT-Action, it is an error for the entire FT-Action and *all* the processes taking part in the FT-Action must cooperatively invoke appropriate recovery measures.

Nesting of FT-Actions: The FT-Actions may be nested. Only strict nesting is permitted. Nesting is needed to program nested recovery techniques, and if recovery is unsuccessful in a FT-Action, it may be attempted in the enclosing FT-Action.

As a practical point, an implementation ought to detect and allow recovery from a *deserter process* [Kim 82]. In terms of CSP, this can occur if an input or output command is never matched or a process dies. This may also occur if a process which is expected to participate in an FT-Action does not. Recovery from this form of exception is especially difficult to implement in a message passing system since a process cannot unilaterally observe the state of another process (systems permitting shared data allow a simple solution).

5.3. Error Recovery with the FT-Action

In this section, we give the notation for a basic FT-Action and describe the primitives needed for the FT-Action. We outline how backward, forward and combined recovery may be implemented using the FT-Action primitives. The primitives have CSP implementations which are described in the following section.

5.3.1. FT-Action Primitives

Each process taking part in an FT-Action must declare an FT-Action entry statement. The entry statement is identified by a name and includes a list of all the other

processes which will also participate in the FT-Action. The name and lists supplied by each of the processes taking part in the FT-Action are compared at run-time to ensure consistency. The syntax for an FT-Action shown in Fig. 5.1.

```
P1 ::[  ...
        FT-Action A with (P2,P3,...,Pn)
                <code>
                exit unless <e>
                <code>
                exit unless <e>
                ...
        end
]
```

Figure 5.1: The FT-Action.

The FT-Action synchronizes recovery schemes involving the processes $P_1, P_2, \cdots, P_n$. Each process should declare a similar statement to that declared in P_1. The body of the FT-Action includes "exit" statements, each of which corresponds to a test point within a test line. When a process reaches an exit, it waits for the other processes in the FT-Action to reach their corresponding exits. The exception "e" of a test line is evaluated by an interprocess voting scheme (described in the next section). This scheme combines exceptions detected by the processes in the FT-Action using exception resolution[Campbell & Randell 83] and distributes the resulting exception value to each process. Thus, the test line returns the same exception to each process. In the case that no process detects an exception, the scheme produces a null value. If the exception is null, then the FT-Action is successful and the exit statement terminates the structure. Otherwise, the processes continue in the FT-Action and recovery measures are invoked. An FT-Action can terminate abnormally by "signaling" an exception.

To ensure the atomicity of the FT-Actions, for the duration of the FT-Action, the process P_1 only communicates with the processes mentioned in its entry statement of the

FT-Action. That is, within the body of the FT-Action, an input or output command in P_1 may only have a process $P_2,P_3,...,P_n$ as the source or destination process, respectively.

In general, implementation of either the forward or backward error recovery scheme will require the use of several exit primitives. In the rest of the section we will omit the body of the process P_1 enclosing the FT-Action.

5.3.2. Backward Recovery

Informally, the FT-Action may be used to specify backward error recovery as shown in Fig. 5.2.

```
FT-Action A with (P2,P3,...,Pn)
        ensure <acceptance test>
        by     <primary>
        else by <alternate>
                ...
        else by <alternate>
        else signal error
end
```

Figure 5.2: Backward Recovery Structure.

The backward error recovery constitutes a Conversation involving the processes $P_1,P_2,...,P_n$. Each process executes its primary and may communicate with other processes executing their primaries in the FT-Action. The process then evaluates its acceptance test. An exception is raised if the acceptance test fails or a run-time error is detected. If no exception is raised, the FT-Action terminates. If an exception is raised by any of the processes, then every process invokes backward error recovery. The FT-Action requires the processes to have the same number of alternates. The FT-Action synchronizes execution of the alternates so that each process keeps in step. The processes may

communicate with one another during the execution of an alternate.

The specification can be transformed into the FT-Action primitives shown in Fig. 5.3.

```
FT-Action A with (P1,P2, ··· ,Pn)
        <save state>
        <primary; acceptance test>
        exit unless <e>
        <restore state>
        <alternate; acceptance test>
        ...
        exit unless <e>
        <restore state>
        signal error
end
```

Figure 5.3: FT-Action Backward Recovery.

The state (values of the variables) of each process is saved after it enters the FT-Action. The saved states of all the processes together form the recovery line. Before the first test line (first set of exit statements), each process evaluates its acceptance test to detect exceptions. If one or more processes detect exceptions, the exception "e" returned by the test line will not be null and the exit statements will not terminate the construct. Instead, the processes roll back and execute the next alternates. After reevaluating their acceptance tests, the processes reach another test line. This sequence is repeated until either the exception returned by a test line is null or the last alternates are attempted. The last alternates are used to "signal" an exception to indicate that the FT-Action has failed.

5.3.3. Forward Recovery

To specify forward error recovery using a FT-Action, the notation presented in[Cristian 82] is used with some modification. Informally, the FT-Action may be used to specify forward error recovery as shown in Fig. 5.4.

```
FT-Action A with (P2,P3,...,Pn)
        <algorithm>
                [ excep e → handler (e) ]
end
```

Figure 5.4: Forward Recovery Structure.

The FT-Action coordinates forward error recovery for the processes $P_1,P_2,...,P_n$. It terminates if no exceptions are raised during the execution of the algorithms. If an exception is raised by any process, then *all* the processes in C are notified of the exception. Each process will then execute its exception handler ("handler (e)") for that exception. Forward error recovery completes when every process either "signals" an exception or successfully completes its handler. If an exception is "signaled" in an FT-Action, the FT-Action terminates abnormally with an exception and, if it is nested within another FT-Action, the exception is raised in the containing FT-Action.

The forward error recovery can be translated into the primitives shown in Fig. 5.5.

```
FT-Action A with (P2,P3, · · · ,Pn)
        <primary>
        exit unless <e>
        [
                e = my_signal          → signal e
                □ e ≠ my_signal → handler(e)
        ]
        exit unless <e>
        [
                e = my_signal          → signal e
                □ e ≠ my_signal → signal error
        ]
end
```

Figure 5.5: FT-Action Forward Recovery.

The first test line after the primaries resolves any raised or signaled exceptions during the execution of the primaries. For simplicity, we require that no exception may be both raised and signaled in the same FT-Action. If the resolution scheme is applied to a raised and a signaled exception, this restriction guarantees that the scheme will not return the signaled exception.

If the exception "e" returned by the test line is null, the exit statement terminates the FT-Action. If each process locally signaled "e", the FT-Action terminates by "signaling" "e". Otherwise, each process attempts recovery by executing the handler for "e".

When the processes complete their handlers, the exception "e" for the second test line is determined. If "e" is a null exception, the exit statement terminates the FT-Action. If each process locally signaled "e" within its handler, the FT-Action terminates by "signaling" "e". Otherwise, the FT-Action "signals" error.

5.3.4. Combined Recovery

Forward and backward error recovery schemes may be combined. One method of using the two techniques in a complimentary manner is shown in figure 5.6.

```
FT-Action A with (P2,P3, · · · ,Pn)
        ensure <acceptance test>
        by <primary>
                [ excep e → handler (e) ]
        else by <alternate>
                ...
        else by <alternate>
        else signal error
end
```

Figure 5.6: Combined recovery structure.

In figure 5.6, a forward recovery scheme is associated with the primary algorithm and would be invoked for the specified exceptions. The backward recovery scheme would be invoked for other exceptions and any exceptions that might occur in the handler. There are many ways to combine forward and backward recovery schemes. Such combinations may be transformed into primitives as before. The translation shown in Fig. 5.7 is a translation supporting the structure described above using the FT-Action primitives.

```
FT-Action A with (P2,P3, ··· ,Pn)
        <save state>
        <primary; acceptance test>
        exit unless <e>
        [
                e = my_signal → signal e
                □ e ≠ my_signal → handler(e)
        ]
        <acceptance test>
        exit unless <e>
        <restore state>
        <alternate; acceptance test>
        ...
        exit unless <e>
        <restore state>
        signal error
end
```

Figure 5.7: FT-Action Forward and Backward Recovery.

Each process completes its primary and evaluates its acceptance test. If the exception "e" returned by the first test line is the null exception, the exit statement terminates the FT-Action. If each process locally signaled "e", the FT-Action terminates by "signaling" "e". In this case, we assume that the signal is a valid result of the primary and passes the acceptance test. Otherwise, each process attempts recovery by executing the handler for "e".

The exception for the second test line is determined after the reevaluation of the acceptance tests. This time, if the exception is not null, backward error recovery is applied and the processes execute their next alternates. Although the FT-Action does not impose any implementation restriction, we choose to simplify recovery strategies by transforming any exceptions which are signaled from a handler into an error.

5.3.5. An Example

To illustrate the use of FT-Actions, we present a simple example. There are two processes, P and Q. P computes n different values, and sends the computed data to Q. The purpose of Q is to construct records using the data sent by P, to sort the records, and then to store them in a file. Two nested FT-Actions provide fault tolerance. FT-Action *A* encompasses the whole activity and provides fault tolerance based on backward recovery. The FT-Action *B* is nested within *A*, and supports forward recovery for the construction of each record.

The backward recovery scheme for Q employs two different methods to produce a sorted file of records. The primary method inputs data, constructs a record, and inserts it in its proper place in a sorted file. The alternate method inputs data, constructs a record, and appends it to an unsorted file. After all the data has been received, the file is sorted. The backward recovery scheme for P uses the same algorithm for both the primary method and the alternate method.

The process Q may receive erroneous data, or might raise an exception in constructing the record. We refer to this exception as a check-sum-error exception. The FT-Action *B* provides forward recovery for this exception. The example is shown in fig. 5.8.

```
P :: [ ....
     FT-Action A with (Q)
        ensure (true) by
        [    i := 0;
             *[ i < n →
                  i := i+1;
                  FT-Action B with (Q)
                       [    Compute (data);
                            Q ! data;
                       ]
                            [excep check-sum-error →
                                 Q ! data;
                            ]
                  end
             ]
```

```
        ]
        else by
        [   i := 0;
            *[ i < n →
                i := i+1;
                Compute (data)
                Q ! data;
            ]
        ]
        else signal error
    end
    ....
]


Q :: [ ....
    FT-Action A with (P)
        ensure (file sorted) by
        [   i := 0;
            *[ i < n →
                i := i+1;
                FT-Action B with (P)
                    [   P ? data;
                        Compute rec using data;
                    ]
                        [excep check-sum-error →
                            P ? data;
                            Compute rec using data;
                        ]
                end
                Insert rec in the file;
            ]
        ]
        else by
        [
            i := 0;
            *[ i < n →
                i := i+1;
                P ? data;
                Compute rec using data;
                Store rec in file;
            ]
            Sort the file;
        ]
        else signal error
    end
    ....
]
```

Figure 5.8: An example using nested FT-Actions

The example simplifies discussion of the following points. In the FT-Action B, though Q detects the exception, P must also take part in the recovery. Consequently, the exception can be regarded as applying to the FT-Action B, and it is irrelevant which process detects the error. Accordingly, it is not possible to determine the process which detects the error by examining the program. Similarly, in the FT-Action A, even though the 'acceptance test' in process P is trivially satisfied, P will have to perform recovery if the acceptance test in the process Q for the FT-Action B fails.

The primaries and alternates do not need to have a similar structure. In the above example, the primaries of the FT-Action A have a nested FT-Action B, while there are no nested actions for the alternates of action A.

Fault tolerance provisions may lead to a loss of concurrency. In the above example, the alternates for the FT-Action A allow P and Q to compute the data and the record concurrently. However, there is less concurrency possible in the primaries of A because of the action B nested within A. The exit of the FT-action B imposes an additional synchronization constraint on the processes P and Q.

5.4. Implementation

An implementation of the FT-Action primitives is described in this section. The implementation uses only the CSP primitives for communication and synchronization between processes. The reliability of the recovery schemes is enhanced by compile and run-time checking.

A combination of compile and run-time checking is used to prevent information smuggling. A syntactic check ensures that, inside an FT-Action, a process only communicates to the other processes named in the entry statement of the FT-Action. A further

run-time check must be used to ensure that the C-Sets of the processes involved in a particular FT-Action are the same, where the C-Set of a process for an FT-Action is the set containing the name of the conversation, the name of the process and the name of the processes specified in the FT-Action statement.

The correct nesting of FT-Actions can be checked at compile-time by examining each process. Each process identifier that occurs in the statement of a nested FT-Action must also occur in the statement of any enclosing FT-Action.

An FT-Action can be transformed into CSP primitives by a preprocessor. For the purposes of implementation, we require the processes within an FT-Action to have a static ordering (for example, we use the lexicographic ordering defined by their identifiers).

5.4.1. FT-Action Entry

Entry of a process into an FT-Action requires synchronization and a C-Set consistency check. The consistency check uses a voting technique based on the Two Phase Commit protocol[Gray 78] Voting is implemented by passing a message up and down a chain of the processes attempting to enter the FT-Action.

The processes whose identifiers are included in the C-Set of an FT-Action are organized into a chain using their static ordering. In a vote, starting from the head of the chain, each process passes C-Set information to its successor. If the C-Set of any process does not agree with the information that the process receives, a C-Set exception is passed on. This ensures that the tail process will receive a C-Set exception if the C-Sets are not consistent. Next, the tail process returns the result of the vote back down the chain to the head. In this way, every process receives an exception if the C-Sets are inconsistent.

If the C-Sets are inconsistent, the FT-Action is aborted by each process.

The voting algorithm is shown in Fig. 5.9. Different algorithms are used for the head, middle and the tail of the chain. Since the chain is constructed using the static ordering of the processes, a compile-time algorithm can construct the voting scheme. We assume that process P_i is the predecessor of process P_{i+1}.

For the head of the chain (process P_1):

P_2 ! C_Set;
[P_2 ? success () → skip
□ P_2 ? failure () → ABORT
]

For the middle of the chain (process P_i):

P_i ? C_Set ;
[(C_Set = My_C_Set) → P_{i+1} ! C_Set
□ (C_Set ≠ My_C_Set) → P_{i+1} ! C_Set_Exception
]
[P_{i+1} ? success () → P_{i-1} ! success ();
skip
□ P_{i+1} ? failure () → P_{i-1} ! failure ();
ABORT
]

For the tail (process P_n):

P_{n-1} ? C_Set;
[(C_Set = My_C_Set) → P_{n-1} ! success () ;
skip;
□ (C_Set ≠ My_C_Set) → P_{n-1} ! failure () ;
ABORT
]

Figure 5.9: Translation of the entry statement

5.4.2. The Exit Statement

The exit primitive is used to terminate an FT-Action if it is successful. The implementation of the exit primitive also uses a chain-based voting scheme to decide whether an exception has been detected by any of the processes in the FT-Action. If an exception is detected, all the processes in the FT-Action must participate in recovery. Each process resolves any exception it may have received from a predecessor process with any exception it has raised and sends the result to its successor process. The final result is sent to each process in the FT-Action by transmitting it back down the chain. The "value" of an exception is taken to be null if no exception occurred. The implementation scheme is shown in Fig. 5.10.

For the head of the chain (process P_1):

P_2 ! my_exception;
P_2 ? final_exception →
    [final_exception == null → exit
    □ final_exception ≠ null → skip;
    ]

For the middle of the chain (process P_i):

P_{i-1} ? exception ;
P_{i+1} ! resolve(exception , my_exception) ;
P_{i+1} ? final_exception →
    P_{i-1} ? final_exception;
    [final_exception = null → exit
    □ final_exception ≠ null → skip
    ]

For the tail (process P_n):

P_{n-1} ? exception ;
final_exception := resolve(exception,my_exception);
P_{n-1} ! final_exception;
[final_exception = null → exit
□ exception ≠ null → skip
]

Figure 5.10: Translation of the exit statement

5.4.3. The Exception Mechanism

So far we have considered planned error detection, that is, errors are detected at certain specified points, which, in the case of the FT-Action, are at the end of the primary or the alternates. However, a process may raise an exception in a point other than at the end of some computation. In this situation, the processes in the FT-Action should not continue with the normal computation. Instead all the processes should execute the exit statement and start the voting process. This also happens if an FT-Action terminates abnormally by signaling an exception, in which case the recovery action of the enclosing

FT-Action should be invoked.

Because of the synchronous message passing scheme of CSP, it is not possible simply to discontinue the normal computation of the process which detects the error. Other processes which communicate in a normal manner with this process will wait indefinitely since the corresponding input or output command will not be executed by the process.

A mechanism is required by which all the processes are notified of the occurrence of the exception. On being notified of the exception, a process should then start voting. The mechanism should also be capable of handling the occurrence of multiple exceptions.

There are several ways to implement such a mechanism. For the purposes of this paper, we propose a simple scheme for implementing such a mechanism which only uses CSP primitives and requires a *broadcaster process* (BP) for each FT-Action. However, our scheme does require output commands in guards. A process that detects an exception communicates with the broadcaster process. The broadcaster process informs other processes taking part in the FT-Action that an exception has occurred. We briefly describe how such a mechanism may be implemented.

The broadcaster process has two phases. In the first phase, it waits for input from any of the processes in the FT-Action. Any process which detects an exception outputs an appropriate message to the broadcaster process. In the second phase, the broadcaster process tries to inform the other processes taking part in the FT-Action that an exception has been detected and will accept further exception messages. In this scheme, the broadcaster process only informs the processes that an exception has occurred. The identity of the exception is still transmitted to the processes by the voting scheme. The broadcaster process is described in fig. 5.11.

```
[
    (i:1..n) P_i ? excep() → skip
]
*[
    (i:1..n) P_i ? excep() → skip
    □ (i:1..n) P_i ! excep() → skip
]
```

Figure 5.11: The Broadcaster Process (BP)

The processes taking part in the FT-Action have to be able to input an exception from the broadcaster process as well as input and output to other processes. Thus, each input or output command of a process is transformed into an alternative command which may also input an exception message from the broadcaster process. If it receives an exception message, the process discontinues normal processing and starts the voting process. Otherwise it continues with the normal processing. The translation of an input or output command C in a process is shown in figure 5.12.

```
[
    BP ? excep() → start voting;
    □ C → skip;
]
```

Figure 5.12: Translation of a command C.

A simple argument reveals the correctness of this scheme. If no exception occurs in the FT-Action, the command C will always be executed by all the processes as the broadcaster process will not be trying to output any exception message. Hence, the FT-Action will execute normally. If an exception occurs, a process which has not yet encountered the exception may reach the exit of the FT-Action, start voting, and detect an exception. Alternatively, it may try to communicate with another process which has already encountered the exception and block. In this case, either the other process detected the excep-

tion and informed the broadcaster process or the other process received an exception message from the broadcaster process. As a consequence, the broadcaster process will either enter its second phase or will already be in its second phase. Thus, the process, if blocked, will receive an exception message from the broadcaster process which will allow it to proceed to vote. So, the processes always reach the voting phase, and the voting phase will then ensure that they are notified of the exception.

Note that the solution involves non-deterministic communication. A process, in an attempt to execute a command C, may non-deterministically either execute C or may input a message from the broadcaster process. Thus, it cannot be exactly predicted when or if an individual process will receive an exception message from the broadcaster process. However, it can be guaranteed that all the processes eventually reach the voting phase and the voting phase ensures that all the processes are informed of the exception.

The proposed scheme is simple and demonstrates that the exception mechanism can be implemented in the framework of CSP and FT-Actions. Other schemes can be designed with and without a broadcasting process. Similarly, other schemes can also be devised for implementing the entry and exit statements.

5.4.4. The Timeout Mechanism

The translation scheme for the entry and exit statements has no mechanism to cope with the problem of a deserter process or the problem of a faulty design of the FT-Action mechanism. If a process is in the C-Set of a set of processes taking part in an FT-Action but it does not itself contain an appropriate FT-Action construct (a deserter process), then it will block the other processes from entering the FT-Action because its neighbors in the FT-Action voting chain will never be able to satisfy their I/O requests during the

first phase of the voting. A similar situation can arise if two processes have different C-Sets for the same FT-Action. There appears to be no satisfactory solution to this problem unless a timeout mechanism is provided in CSP. We describe a simple timeout mechanism which can be employed to solve the problem.

Each process starts a preset timer when it tries to communicate to its successor or predecessor process during the first phase of the voting process. This can be easily implemented using the CSP notation, by a simple translation. If the I/O command to be executed is C, its translation is:

```
[
    C  →  skip
    □ timer = timeout  →  ABORT
]
```

If a matching input/output command to the command C is not executed within the set time, the process locally aborts the FT-Action. The timeout scheme assumes that output commands are allowed in the guards. It is possible to provide translation of a command C without using output commands in the guards, but such a mechanism will not be as simple and natural as the one given above.

If there is a deserter process, then the first phase of voting cannot complete, and some process will timeout. A simple argument shows that if one process taking part in a FT-Action A times out, then all the processes for A will timeout and abort the FT-Action.

Suppose P_i is the first process to timeout. All the processes below P_i in the chain will timeout in the second phase of voting because no message will be sent by P_i to P_{i-1}. The processes above P_i in the chain will time out in the first phase since P_i will not send any vote to P_{i+1}. Hence, all the processes will eventually timeout.

In the simple scheme described here, all the processes in a FT-Action timeout individually. It is possible to make the timeout scheme more efficient by having a mechanism that informs other processes whenever one process times out. Such a scheme is likely to be more complex.

5.5. Discussion

In this chapter we have proposed a notation to specify an atomic action for supporting fault tolerance in a system of Communicating Sequential Processes. The atomic action is called an FT-Action and supports both forward and backward error recovery in an uniform manner. The control structure of a FT-Action is distributed over the processes taking part in it and is implemented using CSP primitives. The number of communication messages needed to coordinate the S-conversation is O(n), where n is the number of processes taking part in the FT-Action. The minimum number of communications needed is also O(n) since all processes must receive at least one message.

Although we have considered practical support for error recovery in concurrent systems, much further research and development is still required. Because of the static channel naming of CSP we have not been able to devise a simple strategy to detect a deserter process. The use of the scheme in practical systems depends on whether the system designers can construct correct and appropriate algorithms. For example, input/output commands in an FT-Action must only match output/input commands in that FT-Action; failure to comply with this structure cannot be detected at compile time and will result in a communication protocol time-out and abortion of the FT-Action.

We believe that a structure like the FT-Action should be used in concurrent languages to provide both backward and forward error recovery support and encourage

the development of reliable concurrent applications. We have demonstrated the practicality of such an approach by devising a mechanism for CSP which can be transformed into CSP language primitives.

CHAPTER 6

A PROTOCOL TO IMPLEMENT ATOMIC ACTIONS IN DATABASES

In a database system, the database is required to be in a consistent state, satisfying the integrity constraints of the database. A transaction is the unit of consistency, that is, a transaction is the unit of computation which transforms a database from one consistent state to another. Therefore, to maintain consistency, it is required that each transaction appear as an operation which is performed indivisibly on the database, and the different transaction must not appear to overlap. In other words, the consistency requirements of a database require that a transaction must execute atomically. However, a transaction may not be indivisibly executable by the hardware as it may consist of many primitive operations. So, system support is required to ensure the atomicity of transactions. In a database system, the problem of supporting atomicity of transactions is referred to as the concurrency control problem.

For database consistency, actually the transactions should be recoverable atomic actions. However, most concurrency control protocols are intended to provide only the concurrency atomicity (hence the name). The failure atomicity is usually provided by the system, and involves log of the computation and commit protocols. In this chapter we present a new protocol to implement (basic) atomic actions in databases. The protocol called the Delay/Re-Read Protocol, employs a combination of preventive and corrective strategies, to ensure the atomicity of transactions. The protocol is deadlock free, and requires no backup data for its operation. The results of this chapter have been reported in[Mickunas et. al. 84].

6.1. Background

The problem of concurrency control in databases has received a good deal of attention in recent years[Bernstein et. al. 79, Bernstein & Goodman 81, Eswaran et. al. 76, Papadimitriou 79, Thomas 79]. In a database system, a transaction is the unit of consistency. Unrestricted concurrency among database transactions can result in an inconsistent database[Bernstein & Goodman 81, Gray 78]. What is needed is that each transaction executes atomically with respect to other transactions. Concurrency control is the activity of co-ordinating concurrent access to a database by various transactions, such that the actions of one transaction do not interfere with the actions of another and each transaction executes atomically. Eswaran et. al.[Eswaran et. al. 76] proposed a protocol, known as Two Phase Locking, to preserve database consistency.

Two Phase Locking requires that each transaction lock the entity it is going to access. A transaction may request a Read lock or an Update lock on an entity. A lock is "granted" only if no other transaction holds a conflicting lock. Furthermore, each transaction runs through both a "growing phase" and a "shrinking phase". In the growing phase a transaction collects the locks that it requires, and in the shrinking phase it releases them. A transaction cannot request any further locks once it has released any lock. A disadvantage of Two Phase Locking is that deadlock may occur. Deadlock is a major concern in concurrency control[Isloor & Marslang 80, Yannakakis et. al. 79], and usually one or more of the deadlocked transactions must be aborted before processing may proceed. This implies that backup data must be maintained so that if deadlock occurs, transactions may be aborted and "undone", thereby restoring the database to a consistent state.

Many variations on locking protocols have been proposed[Bayer & Schkolnick 77, Ellis 80], and it has been demonstrated that locking achieves somewhat better results when the database is structured as a hierarchy[Kedem & Silberschatz 79, Silberschatz & Kedem 82]. The solutions proposed by Thomas[Thomas 79] and Stearns et. al.[Stearns et. al. 76] have been found to be special cases of Two Phase Locking[Bernstein et. al. 79].

The use of locking to maintain consistency is an entirely *preventive measure* that is, it tries to prevent any view of the database from becoming inconsistent. Two Phase Locking assumes the worst case in which synchronization is imposed upon any transaction which *potentially* may conflict with another transaction. Since this is a sufficient but not a necessary condition for actual conflicts[Bernstein & Goodman 79, Bernstein et. al. 79] Two Phase Locking tends to be overly restrictive and results in a reduction in concurrency.

Conflict Graph Analysis[Bernstein et. al. 80] is another technique used to increase the degree of concurrency that employs a preventive technique. It uses a static analysis of the conflict graph to reduce the amount of synchronization needed to ensure that the database remains consistent.

Kung and Robertson[Kung & Robertson 81] proposed a *corrective measure* for concurrency control in an effort to relieve the tight restrictions of locking protocols. In his scheme each transaction works on a private copy of the database and no control is imposed on the actions of any transaction. If, on termination, it is determined that the transaction has operated on a consistent state, the transaction is committed and its changes made permanent. However, if the transaction operated on an inconsistent state, the view of the transaction is "corrected" before its changes are made permanent. The "corrective measure" is to use a backward error recovery technique. The transaction is

aborted and rolled back and then re-executed, in the hope that the transaction will subsequently be presented with a consistent view. In the basic scheme a transaction is prone to repeated abortion. Special measures had to be taken to detect and prevent "starvation" of a transaction.

In the present paper we present a new protocol, which employs both preventive and corrective measures. The method proposed by Garcia-Molina[Garcia-Molina 83] was also a combination of corrective and preventive techniques. The aim of his scheme is to allow nonserializable schedules which preserve consistency and which are acceptable to system users. The technique employs a locking scheme, but adds a corrective technique, to avoid backing up transactions. The method used is to provide *countersteps*, which can be executed to provide *semantic compensation* for the steps performed by a transaction and which need to be "undone". To semantically compensate the activity of a step, a counterstep is needed *for that step.* However, this technique requires the users to provide countersteps to the different steps of the transactions. If the counterstep to a step is not provided, or if it is not possible to construct the counterstep, then the compensation technique cannot be applied.

The protocol, which we call the *Delay/Re-Read Protocol,* acts, on the one hand, in a *corrective* fashion by sometimes forcing a transaction to perform forward error recovery before proceeding with normal processing; it does so upon recognizing that a transaction has read an inconsistent set of data. The error recovery measure requires that the transaction re-read the entities for which it has inconsistent values, and recompute those values for which the inconsistent values were used before it can proceed with normal processing. The protocol acts, on the other hand, in a *preventive* fashion by sometimes imposing a delay before permitting a transaction to write to the database; it does so upon

recognizing that such a write might, at the present time, jeopardize the integrity of the database. A Read request by a transaction is always granted without delay. A Write request may be delayed. The protocol is deadlock-free and no transaction is ever aborted. Consequently, no backup data is needed for the operation of the protocol. The protocol often supports a greater degree of concurrency than Two Phase Locking and no transaction is ever delayed indefinitely.

6.2. System Model

We consider the database to be a collection of distinct objects with unique identifiers, called *entities*. Assertions, called *integrity constraints* specify the possible values of the entities. Integrity constraints govern the possible interactions of *operations* upon entities. A database which satisfies all of the integrity constraints is said to be in a *consistent* state. A complete specification of the integrity constraints for a database might be very large and it might not have an explicit representation.

In order to formalize our moded, we present some definitions.

We denote the set of entities in the database by "E". Each entity may be read or written *indivisibly*.

Definition 6.1. A *transaction,* denoted T^k, is a set of *actions*

$$T^k = \{t_i^k\}_{i=1}^{p_k}$$

together with a linear ordering* $\leq_{T^k}$, on T^k. $<_{T^k}$ is meant to reflect the temporal ordering of the individual actions of T^k. Each t_i^k is a 4-tuple

---

* Recall: a *partial ordering* $\leq$ on a set X is a subset $\leq \subseteq X \times X$ for which $(a,b) \in \leq$ and $(b,a) \in \leq$ implies $a=b$, and for which $(a,b) \in \leq$ and $(b,c) \in \leq$ implies $(a,c) \in \leq$; $(a,b) \in \leq$ is usually written $a \leq b$; if $a \leq b$ and $a \neq b$ then we write $a < b$; $\leq$ is said to be a *linear ordering* on X if for every $a,b \in X$, either $a \leq b$ or $b \leq a$.

$$t_i^k=(k,a_i^k,e_i^k,U_i^k)$$

where

(1) k uniquely identifies the transaction, T^k to which t_i^k belongs

(2) $a_i^k \epsilon \{R,W\}$, called the *operation,* denotes either Read or Write

(3) $e_i^k \epsilon E$ denotes the entity upon which the operation a_i^k is performed

(4) $U_i^k \subseteq 2^E$ (power set of E), called the *Use Set.*

□

In the case $a_i^k=W$, U_i^k denotes a set of entities which are used to compute the new value of e_i^k. Consequently, we may often use a "function" notation when describing a Write action:

$$t_i^k=(k,W,e_i^k(U_i^k))$$

In the case $a_i^k=R$, U_i^k is the empty set. Consequently, we may often omit U_i^k when describing a Read action:

$$t_i^k=(k,R,e_i^k)$$

It is important to note that the Use Set for a particular Write is not necessarily the entire Read Set for the transaction, i.e. a given Write, $t_i^k=(k,W,e_i^k(U_i^k))$ requires only that the entities in U_i^k be consistent. Our protocol derives much of its flexibility from the explicit knowledge of these "data dependencies".

We require that each transaction be *well formed,* that is

(1) a transaction may read an element at most once;

(2) a transaction may write an element at most once;

(3) all Reads of a transaction must precede all of its Writes;

(4) the *Use Set* for a Write action must include the entity being written, i.e. the new value of an entity depends on its old value (among other things), and;

(5) an entity must be read before it can appear in the Use Set of any write.

Formally, these constraints may be written as follows:

Definition 6.2. A transaction $T^k = \{t_i^k\}_{i=1}^{p_k}$ is said to be *well-formed* if and only if the following conditions hold:

(1) if $t_i^k=(k,R,e_i^k)$ and $t_j^k=(k,R,e_j^k)$ then $e_i^k \neq e_j^k$

(2) if $t_i^k=(k,W,e_i^k,U_i^k)$ and $t_j^k=(k,W,e_j^k,U_j^k)$ then $e_i^k \neq e_j^k$

(3) if $t_i^k=(k,R,e_i^k)$ and $t_j^k=(k,a_j^k,e_j^k,U_j^k)$ then either $t_i^k <_{T^k} t_j^k$ or $a_j^k=R$

(4) if $t_i^k=(k,W,e_i^k,U_i^k)$ then e_i^k is in U_i^k

(5) if $t_i^k=(k,W,e_i^k,U_i^k)$ then for every $y \in U_i^k$ there exists t_j^k for which $t_j^k=(k,R,y)$ and $t_j^k <_{T^k} t_i^k$.

□

Our model is thus a generalization of Papadimitriou's "two-step restricted" model[Papadimitriou & Kanellakis 82], in which our restrictions (4) and (5) with $U_i^k=\{e_i^k\}$ reduce to Papadimitriou's broader restriction that an entity must be read before it can be written.

6.3. Consistency

We assume that a given transaction, T^k, transforms the database from one consistent state to another consistent state (although the database may temporarily be in an

inconsistent state while T^k is executing). Our goal is to allow concurrent transactions, yet ensure that when the transactions complete the database will be in a consistent state.

The notion of concurrent transactions is captured by the following definition.

Definition 3.3. Let $T^1, \ldots, T^n$ be transactions. A *schedule*, S, for $T^1, \ldots, T^n$ is the set of actions

$$S = \bigcup_{i=1}^{n} T^i$$

together with a linear ordering, $\leq_S$, on S, for which for all i, $<_{T^i} \subseteq \leq_S$.

□

As before, the relation $<_S$ is meant to reflect temporal ordering (with truly simultaneous actions having an "effective" temporal ordering imposed by $<_S$). Since actions of each transaction are performed in the order the transaction requests them, it follows that if $t_i^k <_{T^k} t_j^k$ then we must have $t_i^k <_S t_j^k$; hence the requirement that $<_{T^k} \subseteq <_S$.

For each transaction T^i, we define its *registration*, (i,w), as a request which precedes T^i's Writes and which follows T^i's Reads. As we shall see, the registration for a transaction will actually be an enumeration of its Write Set. We extend $<_{T^i}$ (and correspondingly, $<_S$) to include (i,w) in the obvious way, viz., $(i,R,x) <_{T^i} (i,w)$ and $(i,w) <_{T^i} (i,W,y)$. Moreover, we further extend $<_S$ so that if (i,w) precedes (j,w) in time, then $(i,w) <_S (j,w)$.

The aim of any concurrency control method is to ensure that the schedules performed on the database transform it from one consistent state to another. Serializability[Eswaran et. al. 76, Papadimitriou 79] has been generally accepted as the

consistency criterion for schedules. Serializability holds that a schedule for transactions $T^1, \ldots, T^n$ is consistent if the state of the database after executing the schedule is the same as it would have been had the transactions been executed one after another in some order. Note that the order (corresponding to some permutation $\{\pi_i\}_{i=1}^n$ *of* $[1,n]$) is not specified.

Given a schedule S, which satisfies the serializability criteria, we refer to the permuted serial execution $T^{\pi_1}, \ldots, T^{\pi_n}$ as an *Equivalent Serial Schedule (ESS).* Such an ESS is not necessarily unique. A schedule having an ESS will be called a *Consistent Schedule.* Not all the schedules are consistent. A *concurrency control protocol* is said to be *consistent* if it ensures that the schedule that finally acts on the database (which might be different from the schedule submitted) is consistent.

Since our model requires that each entity be read before it can be written, a schedule S can be checked for serializability using its corresponding *precedence graph, G_S[Ullman 80].* We construct the graph as follows. The nodes correspond to the transactions. The arcs are determined by the following rule:

If $(i,R,x) <_S (j,W,x)$ or $(i,W,x) <_S (j,W,x)$ or $(i,W,x) <_S (j,R,x)$ for any x, then draw an arc from T^i to T^j.

We note that since $\leq_S$ is a total relation, it follows that the undirected version of G_S is a complete graph.

A schedule S is serializable if its precedence graph is acyclic. It follows that we can find an ESS for S by *topological sorting.*

Clearly the temporal ordering of the registrations induces a serial schedule, $\hat{S}$. If T^i precedes T^j in such a serial schedule, $\hat{S}$, we write $T^i <_{\hat{S}} T^j$. We shall see that the

Delay/Re-Read Protocol, using those registrations, produces a schedule, S whose equivalent serial schedule is $\hat{S}$.

If the registration of a transaction T^i occurs before the registration of the transaction T^j, and the transaction T^i writes the entities before T^j reads them, then the graph G_S can have no arc from T^j to T^i. This is formally stated and proved in the following Lemma.

Lemma 6.1. Let S be a schedule for well-formed transactions. Then the precedence graph G_S has *no* arc from T^j to T^i if $(i,w) <_S (j,w)$, and $(i,W,x) <_S (j,R,x)$ for every (i,W,x) and (j,R,x) in S.

Proof. The proof is by contradiction. There are only three ways that G_S can have an arc from T^j to T^i:

(1) $(j,R,x) <_S (i,W,x)$. Since $<_S$ is anti-symmetric, this directly contradicts the hypothesis that $(j,R,x) \in S$ and $(i,w) <_S (j,w)$ and $(i,W,x) <_S (j,R,x)$.

(2) $(j,W,x) <_S (i,W,x)$. Since T^j is well-formed, we have

$$(j,R,x) <_S (j,W,x)$$

As in case 1, the hypothesis yields

$$(i,W,x) <_S (j,W,x)$$

which, by anti-symmetry of G_S, disallows this case.

(3) $(j,W,x) <_S (i,R,x)$. Since T^i is well-formed, we have

$$(i,R,x) <_S (i,w).$$

Also

$$(j,w) \leq_S (j,W,x),$$

so

$$(j,w) <_S (i,w)$$

which, by anti-symmetry of $<_S$, contradicts the hypothesis that $(i,w) <_S (j,w)$.

□

If the condition in the Lemma is satisfied for every pair of transactions, then G_S will be acyclic. That is, if for every pair of transactions T^i and T^j, if the registration of the transaction T^i occurs before that of T^j implies that T^i writes an entity before T^j reads that entity, then the graph G_S will be acyclic. This if formally stated and proved in the following theorem.

Theorem 6.2. Let S be a schedule for well-formed transactions, and T^i and T^j be any two transactions. Then the precedence graph G_S is *acyclic* if $(i,w) <_S (j,w)$ implies $(i,W,x) <_S (j,R,x)$ for every (i,W,x) and (j,R,x) in S.

Proof. The proof is by contradiction. Suppose that G_S has a cycle involving nodes $T^{i_1}, \cdots T^{i_k}$ $(k>1)$. Since $<_S$ strictly orders the registrations of the transactions, there is one registration (i,w) among $\{(i_1,w),...,(i_k,w)\}$ which is "earliest" in time. Now for every *other* transaction, T^j, $j \in \{i_1,...,i_k\}$ $(j \neq i)$, we have $(i,w) <_S (j,w)$, which by hypothesis implies $(i,W,x) <_S (j,R,x)$. So Lemma 6.1 applies and there can be no arc to T^i from each such T^j, $j \in \{i_1,...,i_k\}$ $(j \neq i)$. Therefore, the presumed cycle involving T^i is not possible.

□

Corollary 6.3. Let S be as in Theorem 6.2 and $\hat{S}$ the serial schedule induced by the registrations. Then S is consistent and $\hat{S}$ is an ESS of S.

Proof. Since by Theorem 6.2, G_S is acyclic, it follows that S is serializable and hence consistent. Moreover, any serial schedule having G_S as its precedence graph is an ESS of

S. Clearly $\hat{S}$ is such a serial schedule.

□

Informally, the theorem lays down a sufficient condition to be satisfied by the schedule that will ensure that every transaction sees a consistent state, that is, the set of values returned by the Reads of the transaction is such that it is the same as the set of values of these entities in some consistent database state. This does not imply that all the Reads must be performed on the same consistent state. A Read can be performed on any database state, possibly transitory and inconsistent, but the set of values read by all Reads must be such that all the values *can co-exist* in some consistent database state. Theorem 6.2 specifies the condition when this is satisfied. This theorem is the basis of Delay/Re-Read Protocol.

In the following sections $W_i(x)$ and $R_i(x)$ mean same as (i,W,x) and (i,R,x) respectively.

6.4. The Delay/Re-Read Protocol

Not all schedules satisfy the condition of Theorem 6.2 in the form they are submitted. The purpose of the Delay/Re-Read Protocol is to control *any schedule* so that the schedule that finally acts on the database satisfies the condition of the theorem.

Each transaction is submitted to a *Transaction Manager* which assigns a *Transaction Process (TP)* to each transaction. A *History File* is used to record the information about the actions performed on the database by the various transactions. This is different from a "log file". A log file, along with data about the actions also records the old and new values of the entities which are modified to provide a "backup". The history file records only a window of activity and no "backup" data is recorded. As we shall see,

the history file need only maintain a record of the actions of recent transactions.

When a transaction requests a Read, the TP permits the read and records this action in the history file. No control is exercised over the Read requests. When a transaction requests a Write, the TP executes the protocol and awaits its instruction(s). The protocol may allow the TP to permit the request or may require the TP to re-read some entities, to re-do the computation, and to re-submit the Write request. When the Write is granted the TP permits the Write and records the action in the history file.

The Delay/Re-Read Protocol is used to ensure that any schedule remains consistent. This is accomplished by a combination of preventive and corrective measures. The Delay/Re-Read Protocol sometimes *delays* a Write request (a preventive action). Alternatively, the Delay/Re-Read Protocol sometimes requires TP to re-read some entities prior to proceeding with a Write (a corrective action), thereby assuring that the Use Set for the Write is consistent.

We assume that the Write Set of the transaction is known by the TP. This information is required after the transaction has performed all of its Reads. A similar assumption has been made in SDD1[Bernstein et. al. 80], and is required in locking protocols in order to determine whether to request a shared or exclusive lock. This does not place any restrictions on what may be read and written by transactions, but rather merely requires that a transaction's Write Set be known. As we will discuss later, the protocol can effectively handle, with possible loss in performance, the situation where the write set is not accurate and contains extra entities. After the transaction has performed all its Reads and before it performs its first Write, it records its Write Set in the history file. If the Write Set of T^i is $\{x,y,z\}$, this is recorded in the history file as $\hat{w}_i(x)\hat{w}_i(y)\hat{w}_i(z)$. The recording of the Write Set is assumed to be an atomic action. This action serves the

purpose of the *registration* as discussed in section 3.

A Read action is recorded as $R_i(entity\text{-}name)$ in the history file. A Write action of the form $W_i(x(U))$ is recorded as $W_i(x)u_i(x_1)u_i(x_2)$ ... $u_i(x_m)$ where each $x_i \in U$. The writing of this sequence is taken to be atomic. (Note that one of the $x_i = x$ and we need not include $u_i(x)$ since it is implied by $W_i(x)$. For the sake of uniformity we will assume that $u_i(x)$ is also recorded)

The protocol consists of three sections. Two sections perform preventive actions, and one section performs the corrective action. The aim is to ensure the consistency by ensuring that the conditions laid down in theorem 6.2 are satisfied. Read requests are granted without delay. The protocol is exercised only for write requests.

Suppose that a write request $W_j(x(U))$ is made by the transaction T^j. To ensure that all the values of the entities in U are "current", the protocol has to ensure that the transaction T^j read those entities after any transaction T^i, which registered before T^j did, has written the entity. This is done in a preventive manner, by waiting for T^i to write the entity. This constitutes the section 2 of the protocol.

If T^j does not have the "current" values of the entities in U, then the corrective action is performed. T^i is forced to re-read those entities that are "outdated", and then recompute the value of $x(U)$ using the latest values. This corrective action is performed only after all transactions which have registered before T^j have performed their corresponding writes. This is the section 3 of the protocol.

However, the possibility of re-reads complicates the situation. Due to the re-reads a transaction may no longer have all its reads before its registration. This implies that even if all the entities in U are "current", $W_j(x(U))$ cannot be permitted until it is cer-

tain that any transaction that registered before T^j will not re-read the entity x. This is done in section 1, and the write is prevented until the occurrence of such a re-read is no longer possible.

Let us now present the Delay/Re-Read Protocol formally.

Let $x,y \in E$

The History File, H is maintained as a string over the alphabet

$$\sum = \{R_i(x), W_i(x), \hat{w}_i(x), u_i(x) \mid x \in E\}$$

Let an ellipses (...) denote an arbitrary string over $\sum$ (possibly of length zero).

Let TP(j) be the transaction process of T^j. The Delay/Re-Read Protocol is shown in figure 1.

Given a request for $W_j(x(U))$,

(SECTION I *)*
1. **for every** $T^i <_S T^j$ **do**
2. { **if** $H = ...R_i(x)...$ & $H \neq ...u_i(x)...$
   then
3. **if there exists** $T^k <_S T^i$ **for which** $H = ...\hat{w}_k(x)...$ & $H \neq ...W_k(x)...R_i(x)...$
4. **then await** $u_i(x)$
   }

(SECTION II *)*
5. **for every** $y \in U$ **do**
6. { **for every** $T^i <_S T^j$ **do**
7. { **if** $H = ...\hat{w}_i(y)...$ & $H \neq ...W_i(y)...$
8. **then await** $W_i(y)$
   }
   }

(SECTION III *)*
9. **if there exists** $y \in U$ & $T^i <_S T^j$ **for which** $H \neq ...W_i(y)...R_j(y)...$
   then
10. {**for every** $y \in U$
11. {**if there exists** $T^i <_S T^j$
12. **for which** $H \neq ...W_i(y)...R_j(y)...$
13. **then** instruct TP(j) to *reread* y
    }
14. instruct TP(j) to *recompute* $x(U)$
15. instruct TP(j) to *resubmit* $W_j(x(U))$
    }
16. **else** *authorize* $W_j(x(U))$.

Figure 1: The Delay/Re-Read Protocol

Sections I and II constitute the preventive action of the protocol (causing delays); section III constitutes the corrective action (causing re-reads).

Informally, the Delay/Re-Read Protocol ensures that there is no arc in G_S from T^j to T^i (where $T^i <_S T^j$). For this the protocol must ensure that for any $x \in E$

(1) $W_i(x) <_S R_j(x)$ (ensured by Section III)

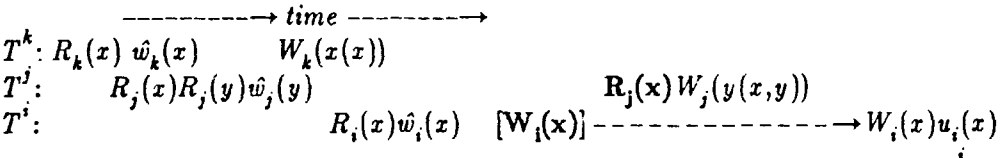
(2) $W_i(x) <_S W_j(x)$ (ensured by Section II)

(3) $R_i(x) <_S W_j(x)$ (ensured by Section I)

Since re-reads are possible, R here means the effective or the final Read. For any $T^i <_S T^j$, if condition 1) does not hold (line 12 and 7), the protocol ensures that T^j waits until $W_i(x)$ is performed (line 8) and then re-reads the entity (line 13), thus ensuring condition 1).

Condition 2) is satisfied since the well-formedness criterion $R_j(x) <_S W_j(x)$ together with Condition 1) ensure that $W_i(x) <_S W_j(x)$.

It takes a bit more thought to see why it is necessary to do anything more to ensure that Condition 3) is satisfied. A transaction, T^i may not perform $W_i(x)$ if some $T^j <_S T^i$ will "soon" be instructed to re-read x. This situation is illustrated by the following time line.



It should be pointed out that in the protocol when we refer to a T^i such that $T^i <_S T^j$, we can exclude from consideration any transaction T^i which terminated before T^j started, since such a T^i automatically satisfies the conditions of Theorem 6.2. To avoid complication, we do not mention it in the protocol.

6.5. Properties of the Protocol

In this section we state and prove a number of properties of the Delay/Re-Read Protocol.

Claim 6.1. The Delay/Re-Read Protocol is consistent.

Sketch of Proof. The above discussion illustrates that for any two transactions T^i and T^j, if $(i,w) <_S (j,w)$ then $R_j(y)$ occurs *after* the $W_i(y)$ that may occur and the final $R_i(x)$ occurs before $W_j(x)$. Thus, the hypotheses of Corollary 6.3 are satisfied, and the resulting schedule is consistent.

□

Claim 6.2. The Delay/Re-Read Protocol is deadlock-free.

Sketch of Proof. T^j is made to wait for T^i only if $(i,w) <_S (j,w)$. Since $<_S$ is a linear ordering, it follows that no deadlock can occur.

⌐

Claim 6.3. For any entity at most one re-read is performed by any transaction.

Sketch of Proof. Before a transaction, T^j discovers in Section III that it must perform a Re-read of some entity y, T^j must first pass through the "gate" of Section II. Section II delays the progress of T^j until all elder transactions, $T^i <_S T^j$ have performed their Writes of entity y. Therefore, upon entry to Section III, transaction T^j is assured that elder transaction have finished their Writes to entity y. Moreover, any *younger* transaction $T^k >_S T^j$ which wants to perform a Write to entity y is delayed in Section I until T^j has performed its Re-read of y.

□

Claim 6.4. No transaction is delayed indefinitely.

Sketch of Proof. Inspection of the protocol shows clearly that no delay is ever imposed on the eldest transaction. Since we assume that transactions always terminate, it follows that eventually every transaction becomes the eldest of the active transactions, and is

therefore immune to further delay.

□

Claim 6.5. If an entity y belongs to the use set of an already authorized write of the transaction T^i, then T^i will not thereafter be requested to reread y.

Sketch of Proof. If a reread is required at all, it is performed when the first write having y in its use set is requested. Else a reread is not necessary. By claim 6.3 there can be at most one reread of y, so subsequent uses of y cannot induce another reread.

□

Claim 6.6. For a schedule, the number of rereads performed by a transaction is fixed and is independent of the use sets.

Sketch of Proof. At the time of registration of a transaction T^i, the first read of the entities by T^i has completed. At registration, for some transaction T^j and entity x, if $(j,w)<(i,w)$ and $H = \cdots \hat{w}_j(x) \cdots$ & $H \neq \cdots W_j(x) \cdots R_i(x)$, then x will be reread by T^i at one of its write requests. Else a reread is not needed. Since there is at most one reread of an entity possible, the number of rereads is fixed.

□

6.6. Discussion

A basic limitation of a preventive approach based on delaying requests is that delaying a request of a transaction also delays the later requests of that transaction, even if there is no actual data dependency between the requests. The actions of a transactions are presented with a linear ordering on them. However, this ordering of actions often does not reflect dependency between the actions. For example, in a transaction contains $R(x)$ followed by $R(y)$ the two reads are ordered but there is no data dependency between the

actions, and they can as well be performed in a different order, without affecting the computation. In the above example delaying $R(x)$ will also delay $R(y)$, even if $R(y)$ could be performed without delay.

A method using only a corrective technique, such as Kung's optimistic approach, avoids this type of unnecessary delay by not delaying any request. However, by not performing a preventive control over any request, transactions have to be restarted in case of conflicts.

The Delay/Re-Read protocol avoids unnecessary 'forced' delays of the read requests, by allowing the reads to take place without any delay, and then rereading some entities when needed. So, the effective ordering of the reads by a transaction is not same as the ordering of the read requests. The cost of breaking the ordering imposed by the ordering of requests in a transaction is extra reads. The protocol maintains the ordering of the write requests.

In the rest of this section we will discuss some issues relating to the operation and the efficiency of the protocol. Two examples are provided and a brief comparison is made with two phase locking.

6.6.1. History File

It may appear that the history string, H, grows without bound. However, there is a simple method by which we can prune H. We observe that we need not record the actions of any transaction that terminated prior to the start of all currently active transactions. Hence, actions of such transactions can be removed from H. We further observe that the performance of a Re-Read, $R_j(x)$, obviates the need for any previous record of $R_j(x)$; thus H can be further pruned of such $R_j(x)$'s.

History file pruning need not be done by the Protocol or the TPs. A background process can maintain and prune H. Since, the record of actions being removed from H are not being considered by the protocol, the background process will not interfere with the protocol, and so no synchronization is needed for it. This technique will keep the history file pruned and make the act invisible to the protocol while reducing its overhead.

6.6.2. Protocol Overhead

We may make a few observations concerning the overhead of the Delay/Re-Read Protocol. Overhead in the Delay/Re-Read Protocol is of three forms:

1) Delay overhead: Caused due to the preventive measures of delaying write requests. Corresponds to the delay of a Write in lines 4 and 8.

2) Re-Read overhead: This is the overhead due the corrective measures of the protocol and includes the overhead of re-computing values. Corresponds to lines 13 and 14.

3) Search overhead: Can be viewed as the execution overhead of the protocol. It is largely caused due to the pattern searching in the history file that is needed for determining the conditions to delay a write or reread an entity.

For a given schedule, the reread overhead is fixed (claim 6.6), and the delay overhead is largely dependent on the accuracy of the use sets. If the use sets are minimal, the delay overhead will be minimal, otherwise extra delays might be incurred (see also discussion below on use sets). Across different schedules these overheads increase as the level of concurrency increases. There is no reread or delay overhead if there is only one active transaction. Moreover, there remains no delay or reread overhead, even with multiple transactions, so long as they operate on disjoint sets of entities. The delay and reread overheads increase only as the interaction among transactions increases, *vis-a-vis* increas-

ingly overlapping Use Sets.

The scheme can also be modified to eliminate the recomputation overhead, where the computation is expensive. Suppose T^i attempts $W_i(x(U))$ where the computation of $x(U)$ is expensive. T^i might view $W_i(x(U))$ as merely a request to write, without first performing the expensive computation. Only once the Write has been authorized, does T^i proceed with the computation of $x(U)$, finally performing the Write.

Though the number of rereads is fixed for a schedule, the cost of rereads can be reduced by proper buffer management. Usually, the permanent database resides on a disc, and there is a database buffer in the main memory, in which the pages that are needed for transaction processing are read[Haerder & Reuter 83]. At the end of a transaction, the changes made by the transaction are made permanent by writing the buffer affected by the transaction to the permanent database. If at the time of reread, the data is still in the buffer the reread will cost considerably less than a regular read, since a regular read requires the data to be first transferred from disc to the buffer. By properly managing the buffers and the times of writing the buffers on the permanent database, it can be arranged the data is still in the buffer when a reread is to be performed. Such a method will considerably reduce the cost of rereading entities.

The search overhead of the protocol can be reduced for a given schedule by using the properties of the protocol and the structure of the history file. The search overhead is reduced by pruning H. It is apparent that if H is pruned, as indicated above, then there is no overhead when there is only one active transaction. The overhead can be further reduced by organizing the history file efficiently. For example, since each pattern the protocol looks for is specified by actions on the same entity, we can divide the history file into sub-history files, one for each entity (or a group of entities). Hashing and/or

indexing can then be used to further reduce the search time.

The overhead can be further reduced by utilizing the claim 6.5 of the protocol. If an entity y has appeared in the use set of an already permitted write of the transaction T^i, then the loops of the sections 2 and 3 need not be executed for the entity y. It follows that once all the entities that are read by T^i have appeared in at the use set of at least one authorized write, then the sections 2 and 3 of protocol actually need not be executed at all for the requests by T^i. However, the cost of reducing the searching of the history file in this manner, is to keep extra information about each transaction.

6.6.3. Use Sets

The Delay/Re-Read Protocol requires explicit knowledge of various data dependencies (via the Use Sets of the Writes). In the present paper we have not addressed the problem of determining such data dependencies. We believe that very often the Use Sets can be specified explicitly by the transaction. Moreover, we believe that even without explicit specification of Use Sets, techniques such as data flow analysis can be used to determine (or to approximate) Use Sets. However, it is clear that the unconstrained problem of automatically determining Use Sets is recursively undecidable.

Though minimal use sets will result in minimal overheads for the protocol, the protocol is robust enough to work properly even if the use set are not accurate. In such a situation it is required that the actual use set be a sub set of the use set used in the protocol. That is, the use set used by the protocol must include at least the entities on which the computation depends, but may contain other entities on which the computation does not actually depend.

The effect of extra entities in the use sets, is a possible increase in the delay overhead. Suppose an extra entity y appears in U of the request $W_i(x(U))$. When the write is requested, two extra actions might be performed, which would not occur if y is not included in U. 1) The protocol might 'await $W_j(y)$' for some transaction T^j; and 2) The protocol might request TP(i) to reread y. Action 2) does not incur any extra overhead, as the reread overhead is fixed; it merely makes the reread occur earlier. However, the delay in 1) would be lesser, and may not even take place, if y is not included in U.

So, the effect of having extra entities in the use sets is to possibly increase the delay overhead, without violating the basic properties of the protocol. Thus, the requirement of accurate use sets can be relaxed, the extreme condition being that the use set of each write request contain all the entities the transaction reads. It should be mentioned that even in this situation the delay overhead might not increase at all, as shown in example 1 below.

Inaccuracy of this nature in the write set can also be handled by the protocol. However, the protocol requires that a write of the entity must be recorded in the history file, if the entity is considered in the write set. This can be easily achieved by performing dummy writes of the extra entities at the end of the transaction (with null use sets). This method also might result in increased delay overhead, as other transactions may have to wait unnecessarily till the dummy write is performed. The reread overhead may also increase, since the dummy writes may induce rereads, where they are not needed. However, the protocol will remain consistent and deadlock free.

6.6.4. Comparison With Locking

It is difficult to compare Two Phase Locking with the Delay/Re-Read protocol because both have different overheads resulting from the different strategies followed. However some comparisons are possible (although we make no attempt here at a full comparison).

1) By using Two Phase Locking transactions can deadlock. The Delay/Re-Read Protocol is deadlock-free. Due to the possibility of deadlock, transactions might have to be aborted to break a deadlock. Transactions are not aborted in the Delay/Re-Read protocol. Furthermore, due to the possibility that transactions may be aborted, the locks of a transaction are usually released at the end of the transactions. If the locks are not held till the end, aborting a transaction may require other transactions to be aborted, and may cause an avalanche of aborts. This restriction further reduces the allowable concurrency with Two Phase Locking.

2) Because of the corrective strategy, the Delay/Re-Read Protocol provides greater concurrency, sometimes at the cost of re-read overhead. But, the Protocol also provides a greater degree of concurrency even without any re-read (see example 1 below). However, there are pathological cases where two phase locking would perform better. Such a case can occur when there is a daisy chain of dependencies but two transactions which are not neighbors in the chain have no dependency between them.

3) The Delay/Re-Read Protocol requires explicit knowledge of various data dependencies, whereas Locking requires no such knowledge.

4) Locking requires a lock table, the size of which is fixed and is a function of the total number of entities in the database. This can be a rather large and unnecessary overhead under low concurrency. Moreover, locking also requires a "log file", so that the

actions of some transactions can be "undone".

The Delay/Re-Read Protocol needs merely the history file, the size of which depends upon the current degree of concurrency. For low concurrency this overhead is low. Moreover, no backup data need be recorded for the protocol.

5) There is a possibility of 'starvation' in Two Phase Locking, when more than one transaction is waiting to lock an entity. The problem is solved by using so-called fair schedulers. No problem of starvation occurs with the Delay/Re-Read Protocol and no extraordinary measures are needed to prevent starvation.

6.6.5. Two Examples

We give two examples. (left-to-right vertical alignment indicates temporal ordering) First in which no re-read is to be done and no write is delayed. In the second example, a re-read is needed.

Example 1:

T^1:$R_1(x)$ $R_1(y)$ $\hat{w}_1(x)\hat{w}_1(y)$ $W_1(x(x,y))$ $W_1(y(y))$

T^2: $R_2(x)R_2(z)$ $\hat{w}_2(z)$ $W_2(z(x,z))$

Two-Phase Locking would force $R_2(x)$ to wait until after $W_1(x(x,y))$, effectively forcing serial execution, while the Delay/Re-Read Protocol permits full concurrency. In this example the ESS produced by the Delay/Re-Read protocol is T^2T^1, while the two phase locking will produce the ESS T^1T^2. This demonstrates that the ESS produced by the Delay/Re-Read protocol and two phase locking may be different. In this example the Delay/Re-Read Protocol will result in optimum throughput (neglecting the search overhead of the protocol). There is no delay or reread overhead in this example. It should also be noted that in this example, if all the use sets were the read sets of the transac-

tions, the delay and reread overhead will still remain the same. (There is only one request, $W_1(y(y))$, whose use set is not the read set. By adding x to its use set will not delay the request, and no reread of x will be needed)

Example 2:

T^1: $R_1(y)R_1(x)$ $\hat{w}_1(x)\hat{w}_1(y)W_1(y(y))$ $W_1(x(x))u_1(x)$

T^2: $R_2(x)$ $R_2(y)\hat{w}_2(x)\hat{w}_2(y)W_2(y(y))$ $\mathbf{R_2(x)}W_2(x(x)$

In this example there is no delay overhead. There is a reread overhead of reading one extra entity. Prior to performing $W_2(x(x))$ T^2 must re-read x (shown emboldened), and must recompute $x(x)$ (if it had already been computed using the old value of x). Locking will force serial execution and a simple minded Two Phase Locking protocol will deadlock.

6.6.6. Concluding Remarks

It has been suggested that there are three basic techniques that can be used for implementing atomic actions in databases- waits, timestamps and rollbacks[Bhargava 82a, Bhargava 82b]. We have shown that there is yet another technique, namely forward error recovery, which can be utilized in place of rollbacks for optimistic protocols. Moreover, this forward recovery is a refinement of rollback inasmuchas it can be used to update only a part of a transactions view, rather than the whole view. Our protocol also demonstrates that the various techniques need not be used in isolation, as they have often been used, but can be combined and used in a complimentary manner. Thus, this approach can exploit the advantages of different techniques while avoiding some of their drawbacks.

The protocol we have presented for implementing atomic actions in a database uses both preventive and corrective measures for ensuring atomicity. The protocol is deadlock-free and accomplishes its "forward recovery" without the need for backup data, without the need for reversing the effects of any Writes, and without aborting transactions. The utility of this method will vary from system to system, depending on the reread overhead in a particular system. We are currently studying the effects of the underlying system structure on the overhead of the protocol.

CHAPTER 7

CONCLUSION AND FUTURE WORK

An atomic action is an activity, possibly consisting of many steps performed by many different processors, that appears primitive and indivisible to any activity outside the atomic action. To other activities, an atomic action is like a primitive operation which transforms the state of the system from one state to another without having any intermediate states. An operation that is executed as an atomic action has the properties of non-interference, non-overlapping and strict sequencing.

Atomicity is fundamental to programming concurrent systems and many different concurrency control schemes which have appeared in many different contexts have actually the same goal: to provide a mechanism that ensures atomicity of system activities. In the recent literature great emphasis has been placed on the database applications of atomic actions. We have shown that the concept of atomicity is more general, provides many additional advantages, and unifies the solutions to many existing problems. Many different concurrency control requirements which appear in different distributed system applications actually have the same goal: to establish the atomicity of operations. Atomic actions are fundamental to the problem of concurrency control in databases, mutual exclusion in operating systems and provision of software fault tolerance.

We have proposed a new model for planned atomic actions, based on transformation sequences of actions. Using this method we have shown that atomic actions are fundamental for fault tolerant software. Many schemes for supporting backward recovery either implement or identify atomic actions in some way. We have shown that an action

that is backward recoverable must be an atomic action. Similarly, we have shown that for general forward recovery in an action, the action must be an atomic action.

These results give a theoretical basis for providing fault tolerant software in terms of a system model based on processes, actions and viable states. It has been made clear that to perform recovery within an action, the action must be atomic. With this as a basis, the problem of providing recovery can be divided into two disjoint problems, the problem of providing atomic actions and the problem of providing recovery. It follows that the continued study of recovery techniques should be discussed within the framework of atomic actions, and the design of recovery techniques should be independent from concerns of atomicity.

We have proposed a notation to specify an atomic actions in a system of Communicating Sequential Processes (CSP). The atomic actions are used for supporting different recovery techniques. In the proposed scheme the planned atomic action is used as the basic unit for providing fault tolerance. The atomic action is called an FT-Action, and both forward and backward error recovery are performed in the context of an FT-Action. An implementation for the FT-Action is proposed, which employs a distributed control, uses CSP primitives, and supports local compile and run-time checking of the forward and backward error recovery schemes.

In databases a transaction is the unit of processing. A transaction is a sequence of read and write actions on the entities of the database. If the actions of different transactions are not properly coordinated an inconsistent database may result. In the database literature, this problem is referred to as the concurrency control problem. What is really desired is that a transaction should appear to execute indivisibly. That is, to maintain the consistency of the database, a transaction should be an atomic action. We have

presented a new protocol, called the Delay/Re-Read Protocol, which uses a combination of preventive and corrective measures for ensuring the atomicity of transactions. The Protocol is deadlock-free, requires no backup data, and often supports a greater degree of concurrency than Two Phase Locking. A transaction is never aborted or delayed indefinitely by the Protocol.

7.1. Future Work

In this thesis we have studied in some depth the relationship between atomic actions and fault tolerant software. The need of atomic actions in databases and operating systems is also well understood. However, even though the concept of atomicity has been in existence for a long time, only recently have people started understanding the fundamental nature of atomic actions. As a result, many consequences of having atomic actions as a basic programming structure are still not well understood or studied. Here we propose a few areas in which we consider that atomic actions can be useful and future work relating to atomic actions can proceed.

7.1.1. Proving Correctness

Almost all the techniques for proving correctness of parallel programs in a shared memory system assume atomicity of actions at some level. The Owicki and Gries method of proving correctness[Owicki & Gries 76b, Owicki & Gries 76a], and the method proposed by Lamport[Lamport 77] both explicitly identify the operations that will be executed atomically. Because these operations are interference free, assertions can be made easily about the results of such operations. If atomic actions are supported by a programming language, then abstract operations can be specified as atomic and guaranteed to be interference free. Such a guarantee about "large" operations would simplify the proof of

a concurrent system considerably.

Let us briefly consider the Owicki and Gries method of proving parallel programs correct[Owicki & Gries 76a, Owicki & Gries 76b] and show that atomic actions will simplify the proofs constructed using their method.

Their technique uses the concept of the "interference" of a process with the *proof* of another. First the proof of each process is studied as an independent, sequential program, disregarding parallel execution. Then, the method requires a proof that the execution of the other processes does not interfere with the proof of each process considered independently. The ability to specify an operation as being atomic in a programming language would simplify the proving of "interference-freeness". An atomic action, being an indivisible operation, could be treated as a single statement free from interference from any other action. So, only the pre-condition and post-condition for the atomic actions need to be interference free. Owicki and Gries also point out that the proof of correctness for an atomic activity is relatively simple[Owicki & Gries 76b]. However, the impact of language supported atomic actions has not been explicitly considered, and remains a research problem.

7.1.2. Correctness of Programs with Exceptions

The problem of proving correctness of sequential programs with exceptions has only recently attracted attention. A proof technique has been developed by Cristian[Cristian 83, Cristian 84]. We expect that atomic actions will provide a convenient framework to extend the technique to concurrent systems.

Cristian's approach considers a sequential program as a set of predicate transformers. One transformer is the standard transformer for normal execution, and the other

transformers are for the different *anticipated* exceptions.

There are two basic underlying assumptions which must be satisfied for this technique to be successful. The first is that only one exception can occur at a time (though different exceptions may occur in different executions), and that the only cause for the occurrence of the exceptional condition is that the computation began in an exceptional domain (in contrast to the *standard domain*). The second assumption is that the execution of the activity cannot be interfered with by the execution of any other activity, so that the predicate transformers of the program are always valid and are independent of the activity of the rest of the system.

In general, both of these assumptions are hard to satisfy in concurrent systems. By definition, an atomic action satisfies the non-interference assumption. The occurrence of multiple exceptions can be resolved into a single exception using the scheme of Campbell and Randell[Campbell & Randell 83]. Therefore, both the basic assumptions in Cristian's scheme can be satisfied by atomic actions. Hence, if atomic actions are used as the basic unit to support exception handling, it should be possible to extend the technique to concurrent systems. However, further work needs to be done to work out the details for this extension.

7.1.3. Program Structuring

In designing concurrent program, some notation is needed to specify the atomicity of operations. In[Dijkstra et. al. 78] the atomic actions assumed are clearly stated. Programming language constructs may be used to specify the atomicity of operations, as in monitors, where an operation on a monitor executes in mutual exclusion. We believe, that provision of atomic actions in programming languages will help the designer in

designing parallel programs, because he can specify an activity as atomic and then be assured of interference freeness of that activity. Consequently, he can concentrate on designing the structure of the system and the activities in the system which should be executed atomically. This would transfer some of the burden of designing parallel programs from the system designer to the language designer and implementor. Moreover, as mentioned in the previous section, reasoning about the programs is expected to be easier and this will aid the rigorous design of concurrent programs. The nesting property of atomic actions supports hierarchical decomposition. The rules for decomposing atomic actions can guide the decomposition of an abstract concurrent system into a hierarchy of atomic actions. This too is an area which needs more research before definite claims can be made.

7.1.4. Decomposition Rules for Atomic Actions

An atomic action can be decomposed into other atomic actions. The recursion implied by this definition terminates when an atomic action can be programmed in terms of primitive atomic actions. To simplify the construction of atomic actions and specifying the semantics of atomic actions, we believe that an atomic action should be decomposed into other atomic actions using a small set of rules. The decomposition rules should aid in design of atomic actions using "smaller" atomic actions, and should aid in verification and specification of semantics of atomic actions using the semantics of the sub atomic actions. Our preliminary results indicate two rules of decomposition.

The set of decomposition rules is based on sequential decomposition and parallel decomposition. A *sequential decomposition* of an atomic action A is a sequence of atomic actions $A_1, A_2, \ldots, A_n$ written:

$$A = A_1; A_2; \cdots; A_n.$$

For all j such that $j \geq 2$ and $j \leq n$, the initial state of atomic action A_j is determined by the final state of atomic action A_{j-1}. The atomic action A terminates when A_n terminates.

An atomic action may be decomposed into a set of atomic actions which can execute concurrently using *parallel decomposition.* An atomic action A, decomposed into parallel atomic actions $A_1, A_2, \cdots, A_n$, can be written as

$$A = A_1 || A_2 || \cdots || A_n.$$

The sub-actions $A_1, \cdots, A_n$ have no defined order of execution. A terminates when *all* the sub-actions terminate. The strict sequence property requires that the final state of A will be same as if $A_1, \cdots, A_n$ were executed in strict sequence. However, the equivalent strict sequence order cannot be determined *apriori* and is non-deterministic, and may result in different final states for A.

Although we have not worked out the details, the atomic actions that constitute a sequential or a parallel decomposition of an atomic action may depend upon more complex control flow mechanisms that permit conditional execution and iterative execution. For example, an iterative atomic action may specify repeated execution of one atomic action or may specify the concurrent execution of multiple instances of one atomic action. Further work needs to be done to demonstrate the completeness and power of these rules.

7.1.5. Independence of Actions and Parallelism

Nested atomic actions can be very useful in exploiting and specifying concurrency and parallelism in a program, especially with the concept of independent actions. Let A

be an atomic action with two atomic actions A_1 and A_2 nested within it.

The two actions A_1 and A_2 are independent if

(1) A_1 and A_2 are parallel atomic actions.

(2) the final state of A is independent of the order in which A_1 and A_2 occur.

If A_1 and A_2 are independent then the result produced by A is the same irrespective of the effective order in which the two nested atomic actions perform their computations. This is in contrast to the situation where the two atomic actions are not independent. In such situations, the final state may be determined in a non-deterministic manner and may depend on the order in which the nested actions occur.

Independent actions are useful in exploiting parallelism in a program, though ensuring independence of actions may be difficult. If two actions are independent, they can be computed separately, possibly on different machines, without affecting the result of the overall computation of the enclosing action. For vector machines the enclosing atomic action is often a loop, and the different sub-actions are the computation of a loop body for different loop indexes. The synchronization needed in case of independent actions is also minimal because of the independence. Synchronization is only needed to determine the final boundary of the enclosing action.

To ensure the independence of sub-actions may not be easy, and work needs to be done to specify rules to easily check for independence and structure atomic actions in such a way that independence of sub-actions is guaranteed. The program transformation techniques[Kuck 81, Padua et. al. 80] aim to restructure loops in such a way that independence of sub-actions is guaranteed.

REFERENCES

Pankaj Jalote was born in Lucknow, India. In 1980, he recieved the Bachelor of Technology in Electrical Engineering from Indian Institute of Technology, Kanpur, India. He recieved the Master of Science degree in Computer Science in 1982 from Pennsylvania State University, University Park, Pennsylvania. In 1985 he recieved his Ph.D. in Computer Science from University of Illinois at Urbana-Champaign. His research interests include fault tolerant software, programming languages, distributed systems and databases, and software engineering.

APPENDIX E

MEDIATORS: A Synchronization Mechanism

Judith E. Grass

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

E

PRELIMINARY: Do NOT Distribute

MEDIATORS: A SYNCHRONIZATION MECHANISM

Judith E. Grass

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

1. Introduction

This paper introduces the *mediator* construct for implementing synchronization and scheduling in distributed systems. This language construct supports systems programming applications that require complex and flexible synchronization and scheduling schemes. The research was prompted by the recognition that many of the existing language constructs either overly constrain concurrency, make expression of some kinds of synchronization and scheduling difficult, or due to formal language design considerations fail to provide practical support for real programmers. The discussion of design goals that follows ndicates examples of each of these failings.

1.1. The Problem

The development of the *mediator* was motivated by the lack of synchronization and scheduling tools to adequately support the development of distributed systems, such as those embedded in space craft. Such tools must meet a number of requirements, including support for modular and structured system design, flexibility, expressiveness, clarity and ease of use.

Modular design is a powerful aid to structuring software development which affects all phases of the software life cycle from specification, through development, testing and validation to maintenance. These three aspects of modularity must be considered: resource modularity, encapsulation of concurrency and synchronization modularity.

Resource modularity is a basic concern in both sequential and concurrent program design. The development of abstract data types and object-oriented programming are an expression of this concern [1]. The encapsulation of data and controlled access to that data through carefully defined operations provide the user with a higher-level, abstract view of a data resource. At the same time, the data is protected from invalid accesses. The module also creates a locality of reference, placing the data and operation definition in one place rather than scattered throughout the code.

Early synchronization tools, such as busy-waits, semaphores [2], and conditional critical regions [3-5], did not create a locality of reference, and so made structuring synchronization difficult. Most recent proposals have recognized this problem, and have taken some version of the abstract data type as a base. In some cases the module is a passive and takes no action until called on by an active process (e. g. monitors [6]). Passive synchronization modules are the rule in constructs based on shared data. Usually constructs based on message passing use an active module. Ada [7], Distributed Processes (DP) [8], Synchronizing Resources (SR) [9], and Argus [10] belong in this category. CSP [11] also belongs to this group, although it is less clearly based on an abstract data type model. In CSP, individual processes encapsulate data. Other processes may access the encapsulated data only by an exchange of messages. The process owning the data resource defines all the operations on the data and localizes data access. Synchronization is not as well localized, because the synchronization depends on the "matching" of input and output commands distributed among many processes.

Although there are many synchronization constructs that support resource modularity, relatively few of them permit real concurrency within the synchronization module. Such constructs do not allow encapsulated concurrency. For instance, monitors allow at most one process to be active at a time. In order to allow multiple processes to access a resource simultaneously (as for reader processes in the well-known readers and writers problem [12]), a monitor is used only to implement a pre-read/ post-read

and pre-write/ post-write protocol, which is called before and after a call to an external *read* or *write* routine [6]. There is no assurance that the protocol will be followed. Deadlock or data corruption may result if it is not. The lack of encapsulated concurrency also makes it difficult to nest modules or to otherwise structure concurrency. Structured concurrency is needed to develop atomic action and fault-tolerant systems [10,13-15]. Concurrent Pascal (which is monitor based) [16], DP [8], Ada [7] and CSP [11] all fail to encapsulate concurrency. Argus [10,17], provides encapsulated concurrency, but with severe restrictions to ensure recoverability. In effect, the restrictions eliminate most parallelism. SR [9], Path Pascal (PP) [18] Distributed Path Pascal (DPP) [19], serializers [20], and MCP [21] do allow specification of encapsulated concurrency.

Synchronization modularity refers to the ability to specify synchronization and scheduling constraints separate from the specification of the resource data abstraction. This additional structuring device aids in system development, but also benefits the validation of design and code. Modular synchronization may also make it possible to develop libraries of synchronizers and schedulers. The isolation of timing aspects contributes to real-time programming as well.

Few constructs provide synchronization modularity. Among those are Path Pascal (PP and DPP) [18], sentinel processes [22], and serializers [20]. Serializers are implemented in a LISP environment. Sentinel processes appear to be the imperative language analog. Both combine built-in counters with a queueing primitive to allow modular specification of synchronization. These constructs appear to be well suited to FIFO scheduling problems and variants of the reader/writer problem, but less flexible than desired [23]. Path Pascal encapsulates most synchronization specifications in a path expression. This often provides a high degree of synchronization modularity. The synchronization modularity is lost when conditional synchronization or scheduling is specified. These must be programmed using nested objects. This results in loss of modularity as well as inefficiency due to the implicit scheduling applied at each level of nesting [24,25]. In order to maintain synchronization modularity, synchronization data must be encapsulated. In addition, there must be support for conditional synchronization and scheduling.

For practical embedded distributed systems, it is important not to overly constrain the system implementer in terms of possible synchronization and scheduling. Synchronization schemes that enforce atomic recoverable transactions (such as Argus and Clouds [26,27]) are essentially ruled out, as they severely limit what can be specified.

Other schemes allow more flexibility in what can be specified, but make the expression of some kinds of constraints difficult. As we noted above, Sentinel Processes make FIFO scheduling problems quite easy to specify, but specification of operation sequences is complicated [28]. In Path Pascal it is easy to specify sequences of operations, but implementing scheduling or conditional synchronization is complicated. It should be possible to express constraints in terms of resource history, resource and synchronization state and information about pending requests.

The configuration of concurrent systems raises other questions about flexibility. Many proposed language constructs for writing distributed systems rely on static systems. In DP and Concurrent Pascal [8,16] processes and modules are instantiated at system creation and never terminate. This is not reasonable for real systems that sometimes require on-the-fly reconfiguration, partial reinitialization, or simply need to print a diagnostic message before failing. DP and Concurrent Pascal do not support resource nesting, so all resources are alive for the duration of the system, which is essentially infinite.

Other constructs allow processes and objects to come and go, but are inflexible in other ways. Frequently communication paths are static. CSP is an extreme case of this [11] in which the sender and the receiver of a message need to know each other's name. This feature of CSP makes it impossible to write libraries of services. CSP was meant to be an exercise in input/output rather than a complete language proposal. Some CSP successors, such as OCCAM [29], have attacked this problem by introducing ports. SR [30] has a similar communication problem. Server processes and clients are tied in a one-to-one relationship that is explicit and rigid.

Most synchronization proposals allow servers to honor requests from anonymous clients. This is a flexible arrangement, but occasionally there are cases in which the client's identity must be known.

Some language constructs provide this information (PLITS [31]), but more often it is left up to the implementer. The *mediator* proposal supports dynamic creation and termination of mediators and flexible communication paths. It also provides a means of identifying clients.

1.2. A Proposal

The *mediator* combines several proposals in an attempt to provide a solution to the problems that are outlined above.

1) The Path Pascal object [18] serves as the model for the mediated object. The path expression is replaced by the mediator. The external view of the object should change little.

2) The Path Pascal object initiation block is replaced by initiation and termination code encapsulated in the mediator.

3) The essential control structure within the mediator is an adaptation of Dijkstra's guarded commands [32]. Our adaptation uses *delay semantics* [8] rather than Dijkstra's *abort* semantics.

4) Guards may contain *status tests* to inquire about pending requests, and boolean tests which may refer to data contained in pending requests [11,31].

5) Requests are associated with unique *keys* that allow the mediator to manipulate requests and implement scheduling.

6) The mediator controls request execution by commands allowing coupled and uncoupled client process execution [28]. There is an explicit command to return results to a client.

7) *Parallel guards* are used to multi-program the mediator. Mediator execution is guaranteed atomic between guard evaluations.

Section two explains these features in greater detail, presenting a syntax and examples.

Section 3 discusses the problems of specifying a new language construct. This paper relies on a BNF grammar to specify syntax, and informal descriptions of the semantics supported by examples. Ultimately, a more formal description must be produced. A meta-language description supported by temporal logic axioms will be used.

Section 4 examines implementation aspects. Many of the individual components of the mediator have been implemented in other languages. The main difficulty is combining these in an efficient manner.

The proposal presented here is preliminary. Some features of the syntax and semantics may change as a formal description is developed, and as implementation issues become more central.

2. Concepts and Notations

The descriptions that follow first present a BNF grammar of a portion of the *mediated object* syntax and then an informal semantic description with examples. In this BNF notation, terminal symbols are represented in bold-face. Optional items are enclosed in braces: {}. Items that may repeat zero or more times are surrounded by parenthesis, followed by an asterisk: ()*.

The *mediated object* is one component of a larger language. This paper does not present a complete language. The "host" language is assumed to be similar to Pascal. Not all non-terminals are resolved. The meaning of such non-terminals should be self-evident.

2.1. The Mediated Object

```
object_type     ::= object constant_def_part type_def_part var_def_part
        operation_part mediator_part end object
operation_part  ::= routine (; routine)*
routine         ::= procedure_process_function_decl
        | operation_decl
operation_decl  ::= entry procedure_process_function_decl
mediator_part   ::= mediator constant_def_part type_def_part var_def_part
        procedure_function_def_part {init_part} body body end body
        {term_part} end mediator
init_part       ::= init stt_list end init
term_part       ::= after stt_list end after
```

The structure of the mediated object is very similar to that of the Distributed Path Pascal object [33]. The scope rules are used. Access to the object occurs only through the declared **entry** procedures. The constants, types and variables defined within the object are shared by the object routines. The mediator regulates access to these routines and may contain data and routines not accessible to any external caller. Mediator data usually consists of flags and counters, although it may also include queue structures for scheduling. Mediator routines may include schedulers and service routines. The mediator may access the data encapsulated in the object only during object initiation and termination.

Like Path Pascal objects, the mediated object is a type. A user may create several instantiations of a given object. The mediator initiation code is executed when an object is instantiated. The termination code executes when the body of the mediator terminates.

The example that follows presents a complete mediated object. In other examples, only the mediator will be presented.

```
reader_writer = object
  var RW_data: data_type;

  entry procedure read (parameters);
      local variables and code for a read operation
  end procedure;

  entry procedure write (val parameters);
      local variables and code for a write operation
  end procedure;

  mediator
      var reader_count : integer;
      init
        reader_count := 0;
      end init
      body
        any i in pid:
            cycle
              req(i); job(i).op = write ->
                  when
                    reader_count = 0 ->
                        exec(i);
                        release(i);
                  end when
```

```
            □
            req(i); job(i).op = read ->
                reader_count := reader_count + 1;
                spawn(i);
        end cycle
    //
    any i in pid:
        cycle
          term(i); job(i).op = read ->
                reader_count := reader_count - 1;
                release(i);
        end cycle
  end body
end mediator
end object
```

This example contains many notations that have not yet been explained, but it does illustrate the declaration of object data (*RW_data*), entry routines (*read* and *write*), and local mediator data (*reader_count*). Entry routine parameters are passed by value and by value-result. Reference parameters seriously compromise data encapsulation and are impractical for distributed implementations.

Clients call for mediator services by using a remote procedure call. Once a process has requested the execution of an entry routine, it is blocked until the results return. This makes the semantics of a call on a mediator (remote or local) the same as that on a local routine. For asynchronous services, the user must implement an explicit buffer.

2.2. Basic Mediator Statements

```
stt_list ::= statement (; statement)*
statement      ::= assignment_stt              -- and other basic statements
        | routine_call
        | choice_block
        | break
        | exec( id )              -- id is a "key" identifier
        | spawn( id )             -- id is a "key" identifier
        | release( id )           -- id is a "key" identifier
        | exec( id ) with id      -- second id is a routine
        | spawn( id ) with id     -- second id is a routine
```

This statement definition applies to the mediator body only. The *choice block* contains a guard command, which is explained below. **Break** causes an exit from the current innermost *choice block*. This is a structured exit to the first statement after the block. The alarm_clock object below shows one possible use of **break**. Assignment statements and routine calls within the mediator body may reference local entities only.

Exec, **spawn** and **release** are special service routines. Their parameter is a *key* variable that uniquely identifies a client and its service request. *Keys* are explained in detail below. **Exec** begins *coupled* execution of a requested the entry operation (identified by the *key*). The mediator starts a process to execute the request, but blocks until that process has terminated. For example, in the *reader_writer* object above, the statement *exec(i);* starts a process to execute the *write* operation for client *i*. The mediator blocks during execution. Because of the guard **when** *reader_count = 0* **->** preceding this statement, no other operation process will be active, and *write* will occur in mutual exclusion. On the other hand, **spawn** initiates *uncoupled* execution. The mediator forks off a process to execute an operation and does not wait for it to terminate. It continues executing mediator code. In the *reader_writer*

object, the statement *spawn(i);* forks off a process to execute a *read* operation for client *i*. When the process terminates, the guard *term(i); job(i).op = read* -> becomes true.

The **release** command returns the results of an operation to the client and removes the request from the mediator. This may be invoked only after an **exec** has been completed, or a status test (**term**, see below) reveals that a spawned process has completed. *Reader_writer* contains examples of **release** both after coupled and uncoupled execution. The separate termination test allows synchronization data to be maintained as services complete. **Release** also makes it possible to delay and synchronize the termination and return of results. This can be used to implement a conversation scheme or some forms of fault-tolerance.

The **with** keyword allows the mediator to substitute an equivalent service for one requested. This may be used for managing several identical units of a resource (line printers for example), when the actual choice of the unit allocated makes no difference to the client. It may also be applicable to deadline scheduling. In coupled execution, the mediator blocks until the substitute service completes. In uncoupled execution (**spawn**), the mediator continues processing; when the substitute service completes, the **term** test for the client process becomes true.

2.3. The Choice Block

```
choice_bock     ::= {key} guard_block
key             ::= any id in pid :
        | any id in id :
                -- first id is a variable id, second is a scalar type
guard_block     ::= cycle guard_list end cycle
        | when guard_list end when
guard_list      ::= guard -> stt_list (□ guard -> stt_list)*
guard           ::= bool_equation
        | status
        | status ; bool_equation
        | otherwise
status          ::= req( id )
        | term( id )            -- id is "key" identifier
```

The mediator *choice block* has many similarities to Hoare's CSP guarded commands [11], which in turn can be credited to Dijkstra [32]. The chosen keywords and semantics are closer to the guarded regions of Brinch Hansen's DP [8]. The concept of *key* is related to Hoare's guard command range, but its intent and implementation are different [34]. The similarities and differences will be discussed below.

A guard block is a control statement in which different statement lists are chosen for execution based on the truth value of the associated guards. Because the evaluation of guards is central to this construct, they will be explained first. The guard command will be described after. Keys will be presented last.

Guards are made up of a *status test* and boolean equations. Mediator guard evaluation always result in either a *true* or a *false* value. The special guard **otherwise** is *true* only when all the other guards in the guard command are *false*.

Status tests allow inquiries about pending requests for mediator service. These are requests to initiate an operation (**req**) or to return results after the operation has completed (**term**). For the guard *req(i)* to be true, the list of unserved requests must contain a request from client *i*. Once the guard has been *fired* (it's associated statement list chosen to execute), *req(i)* can not become true again until the service has been completed and the results returned (by *release(i)*). The guard *term(i)* is similar, becoming true when the execution of an operation for client *i* terminates.

A boolean guard paired with a status tests may test the value of a client's request parameters. Each client's request is represented within the mediator by a job descriptor. The descriptor is a variant

record containing fields for a client processor identifier, a time stamp and the name of the operation requested. The operation field serves as a tag for variant fields allowing access to the operations parameters. The descriptor is accessed using the key by indexing on the variable *job*, as in these examples. A job descriptor for the *reader_writer* would have the following structure:

```
record
   pid : client_process_id;
   ts  : time_stamp;
   case op: (read, write) of
        read:  (read parameter list);
        write: (write parameter list);
   end case
end record
```

In the *reader_writer* object, *job(i).op* references the operation field. Boolean guards may also test the value of the mediator's local variables. Boolean guards paired with status tests are not evaluated if the status test is false.

In the following explanation of a guard command, the execution of the guard is considered in isolation, without considering possible interleaving with other parallel guards. The presence of parallel guards introduces delays, but does not affect the semantics of the guard command.

Mediator guard commands are closely related to Brinch Hansen's guarded regions [8]. The mediator process must wait until some guard condition is true, and then execute the associated statement list. A statement list associated with a *true* guard is said to be *enabled*. A guard whose associated statement list has been chosen and started execution is said to have been *fired*.

Nondeterminism enters when more than one guard is enabled. In this case, one guard will be chosen to fire. A mediator implementation must ensure fairness to avoid starvation problems. The mediator can not delay if there are enabled guards.

In a **when** statement, execution delays until some guard is true. Some guard fires and executes its statements list. When the statement list terminates, the **when** statement terminates. A **cycle** statement repeatedly executes these actions. Its execution loops forever unless a **break** statement is executed.

The *delay* semantics of this guard command differs from Dijkstra's original definition and Hoare's adaptation [11,32]. Hoare and Dijkstra's constructs abort the guarded command when no guard is true. This creates a framework that is very nice for formal verification, but results in servers that do not easily implement waiting. Waiting is usually implemented with a busy loop. This is not practical for real resource managers that may spend a lot of time waiting.

Brinch Hansen implements both *delay semantics* in guarded regions and *abort semantics* for guarded commands. The mediator proposal includes only *delay semantics*, because the inclusion of an **otherwise** guard and a **break** statement make the abort semantics redundant. The **otherwise** guard has other applications for implementing background actions and is a useful shorthand for the negation of all other guards. **Break** also has useful applications beyond simply terminating cycle execution under the conditions that Hoare's guarded commands would terminate. These would be necessary even with a second kind of guard command.

Keys are used to identify the client to the mediator, to access job descriptors for guard evaluation and scheduling purposes and to tie clients to specific resources, as in allocator objects. The *key* concept was suggested by Hoare's CSP process range labels [11,34], but their use in mediators is considerably different. Hoare applies ranges to processes to create a finite number of explicitly and contiguously indexed processes. This function of ranges is not included in mediators. Hoare also applies ranges to guarded commands to substitute values within a given range for a bound variable in the guard statements. The following example is from [11]:

(i:1..n)G -> CL stands for

G1 -> CL1 □ G2 -> CL2 □ ... □ Gn-> CLn.

In effect, the guard is expanded by creating a guard and statement list for every value of *i*. The application of ranges in Hoare's guarded commands is quite general.

In the mediator proposal, keys serve only to identify client processes. Like Hoare's ranges, a key statement (**any** ...) defines a key variable which will be bound within the guard command it modifies. Consider the following choice block:

```
any i in range:          range = 1..10
        cycle
                req(i); job(i).op = A ->
                        exec(i);
                        release(i);
                □
                req(i); job(i).op = B ->
                        x := x + 1;
                        exec(i);
                        release(i);
        end cycle
```

It is executed as if it were written:

```
        cycle
                req(1); job(1).op = A ->
                        exec(1);
                        release(1);
                □
                req(1); job(1).op = B ->
                        x := x + 1;
                        exec(1);
                        release(1);
                □
                req(2); job(2).op = A ->
                        exec(2);
                ...

                □
                req(10); job(10). op = B ->
                        x := x + 1;
                        exec(10);
                        release(10);
        end cycle
```

In this example the value of the key identifier is bounded and of the user declared type *range*. Usually the implicit process identifier (the *pid* descriptor field) will be used as the key. The process identifier range is not bounded. Mediator writers do not need to know explicitly what process identifier values are being used, just that they are unique. Although, in an abstract sense, a potentially infinite key variable range implies an infinitely expanded guard, there is no need to implement them that way. Keys are always associated with status tests. If no client with a given process identifier has a request for the mediator, there is no possibility of a *true* status test, so there is no reason to evaluate such a guard. Guard evaluation can be limited to the set of clients with requests. In fact, it can be restricted further when fairness is taken into consideration.

Key variables are tied to job descriptors. For a *pid* type key, the reference is to the process identifier field. In the case of a user declared range type, as in the example above, some other unique field of the descriptor must be used. This will require an explicit declaration in the mediator.

The following mediator for the object *diner* implements synchronization for the dining philosophers problem:

```
mediator
  type range = 0 .. n-1
  var
          fork : array [range] of (free, inuse);
          j    : range;
  Init
      for j := 0 to n-1;   fork[j] := free;
  end Init

  body
      any i in range:
        cycle
            req(i); job(i).op = eat and fork[i] = free
                    and fork[(i+1) mod n] = free ->
              fork[i] := inuse;
              fork[(i+1) mod n] := inuse;
              spawn(i);
            □
            term(i); job(i).op = eat ->
              fork[i] := free;
              fork[(i+1) mod n] := free;
              release(i);
        end cycle
  end body
end mediator
```

The client process executes the statement *diner. eat (rangeprm);* to call the mediator's *eat* routine. This solution will not deadlock, but it is still possible for starvation to occur. A slightly more complex solution using nested mediators solves that problem.

2.4. Parallel Choice Blocks

body ::= choice_block (// choice_block)*

Parallel choice blocks are proposed to allow different sets of guards to be evaluated at different times during mediator execution. It allows the mediator to shuffle together the guards of several guarded commands. The choice of the notation // to separate parallel choice blocks is deliberate. A mediator containing parallel choice blocks uses a multiprogrammed thread of control, one thread of control for each choice block. Only one thread of control is active at a time. The active control block can change only when guards are evaluated. This creates atomic execution of the statement lists between guard evaluations. The mediator body terminates if all of the parallel guard blocks terminate.

Consider the stripped down example that follows. (Labels have been included to make discussion easier).

```
body
  l1: cycle
        A ->    l2: SA;
```

```
                l3: when  B ->   l4: SB  end when
    end cycle
      //
  m1: cycle
        C ->    m2: SC;
                m3: when  D ->   m4: SD  end when
    end cycle
end body
```

In this example, *A, B, C, D* are guards. *SA, SB, SC, SD* are statement lists. The control vector of this mediator has two elements. Initially it is: <l1, m1>. When guard evaluation occurs in the initial state, the guards *A* and *C* are evaluated. Like for isolated guard commands, the associated statement list of some true guard will be executed. If the guard *A* from the cycle l1 is fired, the statement list starting at l2 will begin execution. It will continue executing without interruption until the new guard command at l3 is encountered (assuming SA contains no guard commands). At this point the control vector is <l3, m1>, and the new guard evaluation includes the guards *B* and *C*. Considering all possible combinations, the set of guards evaluated at any one time may be: [A, C], [A, D], [B, C] or [B, D].

The parallel guard notation is an easy and concise way of specifying changing sets of enabling conditions. It is possible to rewrite a parallel guard as one large simple guard command by using a distribution algorithm. The resulting guard command is considerably more bulky and actually less clear.

The introduction of a control vector within the mediator does not create the same complications for reasoning about programs that are usually associated with parallel processes. The control flow in mediators is very restricted, giving recognizable atomic actions. This atomicity of action combined with the small size of mediators and the explicit statement of preconditions in the guards makes it quite easy to reason about the behavior of parallel guards.

Manna and Pneuli have created a formal tool that can be applied to parallel guards. In the paper [35], they apply temporal logic to validating multiprograms in a control framework that is much less restrictive. Their results are applicable to the mediator construct.

The reader/writer mediator demonstrates one application of the parallel guard. In that example, firing the guard *req(i); job(i).op = write* executes the associated statement, which is a *when* statement. As long as its guard *reader_count = 0* is false, the guard can not fire. No new *write* or *read* operations will be accepted, but the second parallel guard will allow *read* operations to finish up and leave the mediator. Parallel choice blocks coupled with nested guard commands gives a convenient way to block some actions while permitting others.

2.5. Some Additional Examples

The examples that follow demonstrate some applications of mediators. Only the mediator portion is included.

2.5.1. Alarm clock

The alarm clock object delays a caller for a time period specified in the call's parameter *n*. Calls to the *wake* operation cause a delay. Calls to the *tick* operation advance the clock.

```
init
  now := 0;
end init
body
  any i in pid:
      cycle
        req(i); job(i).op = wake ->
                    -- start the op, but termination will be delayed
```

```
                job(i). out_time := now + job(i). n;
                spawn(i);
        end cycle
        //
    any i in pid:
        cycle
            req(i); job(i).op = tick ->
                now := now + 1;
                exec(i);
                release(i);
                any j in pid:
                    cycle
                        term(j);  job(j).op = wake
                                and job(j).out_time <= now  ->
                            release(j);
                        □
                        otherwise -> break    -- exit cycle
                    end cycle
        end cycle
end body
```

The field *out_time* must be declared for the operation *wake* job descriptor within the mediator. This example contains the use of **otherwise** and **break** for cycle termination. The incrementation of *now* could also be done internally.

2.5.2. Shortest Job Next

This mediator implements a scheduler that chooses the job with the lowest estimated service time for the next execution. Requests are served in mutual exclusion. This framework is applicable to many scheduling problems.

```
body
  any i in pid:
        cycle
            req(i); job(i).op = service ->
                enqueue (i, job(i).estimate);
        end cycle
        //
   cycle
        queue_not_empty ->
            j := dequeue;
            spawn(j);              -- initiate service operation
            when
                term(j); job(j).op = service ->
                    release(j);
            end when
   end cycle
end body
```

The first guard command simply calls a user defined process local to the mediator to queue up job descriptors in order of their estimate parameter. The second guard command removes the head element of the queue and starts its execution. The *spawn* and wait for termination allows the mediator to continue enqueueing new requests while a service operation is executing.

The key variable *j* in the second guard command is set by direct assignment rather than through a **cycle** nodifier.

2.5.3. An Allocator

An allocator gives a client process exclusive rights to a resource for a series of accesses. The client must request an allocation, then may make repeated calls on the resource. Finally, the client must explicitly release the resource before it can become available to another client.

```
body
  any i in pid:
      cycle
        req(i); job(i). op = allocate ->
              exec(i);
              release(i);
              cycle
                 req(i); job(i).op = use ->
                       exec(i);
                       release(i);
                 □
                 req(i); job(i).op = free ->
                       exec(i);
                       release(i);
                       break;
              end cycle
      end cycle
end body
```

This example uses the key binding of the outer **cycle** to restrict use of the resource to one process in the inner **cycle**. This also demonstrates another use of the **break** statement.

3. Formalisms

The description of mediators presented in this paper is adequate for an introduction to mediator concepts and notations. This overview avoids some of the dirty details that must be faced in implementing and documenting a language. It is inadequate as a complete specification.

There are a great number of possible choices for specifying a programming language or language construct. These include operational definitions (often accomplished by an implementation), attribute grammars, denotational semantic descriptions and various flavors of axiomatic definitions [36]. Because the purpose of this research is to develop a language construct, and not to develop a specification methodology, only those methods that are well developed for specifying concurrency and offer ready-made tools are useful.

Denotational semantics is quite highly developed for application to sequential programming languages. Application to concurrent languages is very much an ongoing field of research [37,38]. For this reason, denotational semantics is not a reasonable specification tool for mediators.

A mediator specification must meet two particular goals. It should give adequate guidance to an implementer and serve as a support for users. The first of these goals may be met by an operational description, the second by an axiomatic description. There are several ways to operationally describe a language, as there are several systems of axiomatic descriptions.

The fact that the mediator does not represent an entire language implies that it must be implemented as an extension of an existing language. Pascal is the most likely host due to its wide availability and because the compiler source is easy to obtain. Implementation will probably be through bootstrapping from a host compiler. This makes it especially attractive to make a meta-language description of

the construct using the host language for a meta-language. This would give a great deal of aid to a potential implementer.

A meta-language description is of some use to a potential user, but not unambiguous enough. Some effort should be made to support verification. The goals of this research limit what can reasonably be done. Once again, a ready-made tool is needed. The current best candidate is temporal logic as described by [35,39]. This framework has been further developed by [40,41]. Temporal logic allows direct reasoning about program control, making most uses of auxiliary variables or history variables unnecessary. The directness of temporal reasoning is very attractive. The fact that Manna and Pneuli directly addressed multiprogramming in their work is also helpful.

Temporal logic provides a well developed logical system, but it does not provide us with the axioms for mediators. These are being developed now. It is not expected that the results will be either complete or consistent. Complete and consistent axiom systems are easier to develop for languages that have been simplified to achieve nice formal characteristics. Mediators are meant to be applied to real problems. Where choices have been made between utility and formal characteristics, utility has been prefered. Formalizing these features may be difficult.

The axiomatic specification of mediators hopefully will be complete enough to gain an insight into how mediators might be validated. It should also be useful as a design tool. There has been a good deal of research into temporal logic as a specification tool [22,28,42,43]. The creation of a temporal axiom system for mediators could enhance their value as a development tool.

4. Implementation

Implementing mediators should not present significant problems, because many of the components of the construct have been implemented in other languages. The main problem will be fitting the components together in an efficient manner.

Several different implementation exist of remote procedure calls for distributed processing. They are implemented in Ada [7], DP [8], and SR [30]. A remote procedure call can be implemented as an exchange of messages between the client and mediator. The client sends a request message containing the name of the operation requested, its process identifier and parameters. It then waits to receive a reply, which will arrive when the mediator has released the operation. The mediator receives a request and creates a job descriptor. This is placed in the list of pending requests, becoming available for status tests. The job descriptor is destroyed when the mediator releases a job and returns results to the client. In the perception of the client process, a remote procedure call appears to be no different than a simple local procedure call.

The **exec** and **spawn** statements require system support to create a process and schedule its execution. McKendry's *execute* statement [44] may be applicable, or these may be implemented using lower level system calls.

Guard command evaluation can be a source of inefficiency, either because guards are constantly being reevaluated or because of the large number of guards that need to be reevaluated each time. The number of guard reevaluations in mediators can be limited because of their limited application. After a guard evaluation, only certain events may change the value of the guards: the arrival of a new request, the termination of an active request or the execution of mediator statements after a guard has fired. If all guards have evaluated as false, there is no need to reevaluate the guards until either new requests arrive, or active requests terminate.

It is possible to limit the number of guards considered during evaluation as well. The evaluation of guards containing status tests can be limited in two ways. Status tests need only be evaluated for clients that are present in the mediators list of pending requests, since the value of any other status guard is automatically *false*. Application of fairness limits the evaluation of status tests for clients as well. These can be evaluated in the order of their arrival until an enabling guard is found.

The evaluation of pure boolean guards can not be limited this way. Fortunately, these are likely to be few in number. These also present a fairness problem. It is easy to apply a fair ordering criteria for

requests based on time of arrival, but such criteria can not be applied to simple boolean guards that may, without firing, become *true* and *false* repeatedly. Implementing fairness may require implementing some kind of counter so that these guards may be ordered. It is important to make the effort to ensure some kind of fairness, as it is nearly impossible to prevent requests from starving otherwise. The solution of making the user responsible for forcing fair execution from an inherently unfair mechanism, (as advocated by Hoare for CSP [11]) ducks the issue.

The design of mediators is best suited to a system made up of distributed multiprocessor nodes, with one or several mediated objects installed at each node. Implementing mediators on such a system should be straightforward. Implementation of mediators on a uniprocessor is also possible using multiprogramming, but would probably be very inefficient. Mediators implemented on a distributed network of uniprocessors could work quite well. This could be accomplished by multiprogramming the mediated object on one node, or by allowing the mediator to exist on one node, and execute operations at remote nodes. The limiting factor would be the amount of object data that would need to be sent to the remote service nodes.

5. Conclusion

This paper has presented a preliminary proposal for a new language construct, the *mediator*, that may serve as a useful tool in programming distributed embedded systems. Mediators allow direct programming of synchronization and scheduling and are able to directly use both information about a pending request and the present synchronization state. This makes mediators a powerful construct for synchronization and scheduling applications.

At the same time, the design of mediators supports structured design of concurrent programs. Mediators also provide some support for program verification.

Finally, mediators should not present significant implementation problems and are adaptable to a number of distributed architectures.

References

1. Dijkstra, E. W. **A Discipline of Programming**. Prentice-Hall, Englewood Cliffs, NJ, 1976.
2. ---. *Cooperating Sequential Processes.* In: **Programming Languages**, F. Genuys, ed. Academic Press, New York, NY, 1968.
3. Hoare, C. A. R. *Towards a Theory of Parallel Programming.* In: **Operating Systems Techniques**, C. A. R. Hoare and R. H. Perrott, ed. Academic Press, London, 1972, pp. 61-71.
4. Brinch Hansen, P. *Structured Multiprogramming.* **CACM** (July, 1972) vol. 15, no. 7, pp. 574-578.
5. ---. *Concurrent Programming Concepts.* **ACM Computing Surveys** (Dec. 1973) vol. 5, no. 4, pp. 223-245.
6. Hoare, C. A. R. *Monitors: An Operating System Structuring Concept.* **CACM** (Oct. 1974) vol. 17, no. 10, pp. 549-557.
7. Defense, U. S. Department of. *Programming Language Ada: Reference Manual.* In: **Vol. 106 Lecture Notes in Computer Science**. Springer-Verlag, New York, NY, 1981.
8. Brinch Hansen, Per. *Distributed Processes: A Concurrent Programming Concept.* **CACM** (Nov. 1978) vol. 21, no. 11, pp. 934-941.
9. Andrews, Gregory R. *Synchronizing Resources.* **ACM TOPLAS** (Oct. 1981) vol. 3, no. 4, pp. 405-430.

10. Liskov, Barbara and Robert Scheifler. *Guardians and Actions: Linguistic Support for Robust, Distributed Programs.* **ACM Tran. Prog. Lang. and Syst.** (July 1983) vol. 5, no. 3, pp. 381-404.

11. Hoare, C. A. R. *Communicating Sequential Processes.* **CACM** (Aug. 1978) vol. 21, no. 8, pp. 666-677.

12. Courtois, P. J., F. Heymans and D. L. Parnas. *Concurrent Control with Readers and Writers.* **CACM** (Oct. 1971) vol. 14, no. 10, pp. 667-668.

13. Campbell, Roy H. and Brian Randell. "Error Recovery in Asynchronous Systems", Tech. Report: Univ. of Illinois, Urbana-Champaign, Dept. Comp. Sci., Urbana, IL, 1984.

14. Jalote, Pankaj and Roy H. Campbell. "Recoverability of Actions and Atomicity", Submitted to a conference. Dept. of Comp. Sci., University of Illinois, Urbana-Champaign, Urbana, IL, 1985.

15. Best, E. and B. Randell. *A Formal Model of Atomicity in Asynchronous Systems.* **Acta Informatica** (1981) vol. 16, pp. 93-129.

16. Brinch Hansen, P. *The Programming Language Concurrent Pascal.* **IEEE TOSE** (1975) vol. SE-1, pp. 199-206.

17. Weihl, William and Barbara Liskov. *Specification and Implementation of Resilient, Atomic Data Types.* **SIGPLAN Notices** (June 1983) vol. 18, no. 6, pp. 53-64.

18. Campbell, R. H. and R. B. Kolstad. *An Overview of PATH PASCAL's Design.* **SIGPLAN Notices** (Sept. 1980) vol. 15, no. 9, pp. 13-14.

19. Kolstad, Robert Bruce. "Distributed Path Pascal: A Language for Programming Coupled Systems", Ph. D. Thesis, Tech. Report: Dept. Comp. Sci., University of Illinois at Urbana-Champaign, UIUCDCS-R-83-1136, Urbana, IL, 1983.

20. Hewitt, C. E., and R. R. Atkinson. *Specifications and Proof Techniques for Serializers.* **IEEE TOSE** (1979) vol. SE-5, no. 1, pp. 10-23.

21. Bahsoun, H., C. Betorne and L. Feraud. *Une Expression de la Synchronisation et de l'Ordonnancement des Processus Concurrents par Variables Partagees.* In: **Proc. 6th Int. Symp. on Programming: LNCS 167**, M. Paul and B. Robinet, ed. Springer-Verlag, New York, NY, 1984, pp. 13-22.

22. Ramamritham, Krithivasan and Robert M. Keller. *Specifying and Proving Properties of Sentinel Processes.* In: **Proc. 5th Int. IEEE/ACM Soft. Eng. Conference.**, 1981, pp. 374-381.

23. Bloom, Toby. *Evaluating Synchronization Mechanisms.* In: **Proc. 7th Symposium on OS Principles (Pacific Grove, CA. Dec 10-12)**. ACM, New York, NY, 1979, pp. 24-32.

24. Grass, Judith Ellen. "Mediators: A Synchronization Mechanism", Prelim. proposal, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 1984.

25. Campbell, Roy H., Jeff Donnelly, Raymond B. Essick, Judith Grass, Dirk Grunwald, Pankaj Jalote and David A. McNabb. "The Embedded Operating System Project: Midyear Report, May 1984", Software Systems Research Group, University of Illinois at Urbana-Champaign, Dept. of Computer Science, Urbana, IL, 1984.

26. Allchin, J. E. and M. S. McKendry. *Synchronization and Recovery of Actions.* **Preprint: 2nd ACM SIGACT-SIGOPS Symposium on Priciples of Distributed Computing** (Aug. 17-19, 1983).

27. Allchin, James E. and Martin S. McKendry. "Support for Objects and Actions in Clouds:

Status Report", Tech. Report Georgia Institute of Technology, Atlanta, GA, Atlanta, GA, 1983.

28. Ramamritham, Krithivasan and Robert M. Keller. *Specification of Synchronizing Processes.* **IEEE TOSE** (Nov. 1983) vol. SE-9, no. 6, pp. 722-733.

29. May, D. *OCCAM.* **SIGPLAN Notices** (April 1983) vol. 18, no. 4, pp. 69-79.

30. Andrews, Gregory R. *"The Distributed Programming Language SR - Mechanisms, Design and Implementation".* **Soft. Pract. and Exper.** (1982) vol. 12, pp. 719-753.

31. Feldman, Jerome A. *High_Level Programming for Distributed Computing.* **CACM** (June 1979) vol. 22, no. 6, pp. 353-367.

32. Dijkstra, Edsger W. *Guarded Commands, Nondeterminacy and Formal Derivation of Programs.* **CACM** (Aug. 1975) vol. 18, no. 8, pp. 453-457.

33. Campbell, R. H. and R. B. Kolstad. *PATH PASCAL User Manual.* **SIGPLAN Notices** (Sept. 1980) vol. 15, no. 9, pp. 15-24.

34. Hoare, C. A. R. *A Model for Communicating Sequential Processes.* In: **On The Construction of Programs**, R. M. McKeag and A. M. McNaughton, ed. Cambridge University Press, Cambridge, UK, 1980, pp. 229-243.

35. Manna, Z. and A. Pneuli. *Verification of Concurrent Programs: The Temporal Framework.* In: **The Correctness Problem in Computer Science**, R. S. Boyer and J. S. Moore, ed. Academic Press. London, UK, 1983, pp. 215-273.

36. Pagan, Frank G. **Formal Specification of Programming Languages: A Panoramic Primer**. Prentice Hall, Inc., Englewood Cliffs, NJ, 1981.

37. ---. *Toward Complete Programming Language Descriptions That Are Both Formal and Understandable.* **Soft.- Pract. and Exper.** (March 1984) vol. 14, no. 3, pp. 199-206.

38. Pneuli, A. *The Temporal Semantics of Concurrent Programs.* In: **Semantics of Concurrent Computation: LNCS 70**. Springer-Verlag, New York, NY, 1979, pp. 1-20.

39. Manna, Z. and A. Pneuli. *Verification of Concurrent Programs: Temporal Proof Principles.* In: **Logics of Programs: LNCS 131**, Dexter Kozen, ed. Springer- Verlag, New York, NY, 1982, pp. 200-252.

40. Owicki, S. S. and L. Lamport. *Proving Liveness Properties of Concurrent Programs.* **ACM TOPLAS** (July 1982) vol. 4, no. 3, pp. 455-495.

41. Hailpern, B. *Verifying Concurrent Processes Using Temporal Logic.* In: **LNCS, Vol. 129**. Springer-Velag, New York, NY, 1981.

42. Lamport, Leslie. *Specifying Concurrent Program Modules.* **ACM TOPLAS** (April 1983) vol. 5, no. 2, pp. 190-222.

43. Manna, Zohar and Pierre Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications.* **ACM TOPLAS** (Jan. 1984) vol. 6, no. 1, pp. 68-93.

44. McKendry, Martin S. and Roy H. Campbell. *Implementing Language Support in High-Level Languages.* **IEEE TOSE** (May 1984) vol. SE-10, no. 3, pp. 227-236.

PRELIMINARY: Do NOT Distribute

MEDIATORS: A SYNCHRONIZATION MECHANISM

Judith E. Grass

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

7.1. Introduction

This paper introduces the *mediator* construct for implementing synchronization and scheduling in distributed systems. This language construct supports systems programming applications that require complex and flexible synchronization and scheduling schemes. The research was prompted by the recognition that many of the existing language constructs either overly constrain concurrency, make expression of some kinds of synchronization and scheduling difficult, or due to formal language design considerations fail to provide practical support for real programmers. The discussion of design goals that follows indicates examples of each of these failings.

7.2. The Problem

The development of the *mediator* was motivated by the lack of synchronization and scheduling tools to adequately support the development of distributed systems, such as those embedded in space craft. Such tools must meet a number of requirements, including support for modular and structured system design, flexibility, expressiveness, clarity and ease of use.

Modular design is a powerful aid to structuring software development which affects all phases of the software life cycle from specification, through development, testing and validation to maintenance. These three aspects of modularity must be considered: resource modularity, encapsulation of concurrency and synchronization modularity.

Resource modularity is a basic concern in both sequential and concurrent program design. The development of abstract data types and object-oriented programming are an expression of this concern [Dijkstra, 1976]. The encapsulation of data and controlled access to that data through carefully defined operations provide the user with a higher-level, abstract view of a data resource. At the same time, the data is protected from invalid accesses. The module also creates a locality of reference, placing the data and operation definition in one place rather than scattered throughout the code.

Early synchronization tools, such as busy-waits, semaphores [Dijkstra cooperating, 1968], and conditional critical regions [Hoare, 1972 Toward a Theory, Brinch Hansen 1972, Brinch_Hansen 1973 Concurrent], did not create a locality of reference, and so

made structuring synchronization difficult. Most recent proposals have recognized this problem, and have taken some version of the abstract data type as a base. In some cases the module is a passive and takes no action until called on by an active process (e. g. monitors [Hoare 1974 Monitors]). Passive synchronization modules are the rule in constructs based on shared data. Usually constructs based on message passing use an active module. Ada [U. S. Department of Defense 1981 Reference Manual], Distributed Processes (DP) [Brinch_Hansen 1978 CACM], Synchronizing Resources (SR) [Andrews 1981 TOPLAS], and Argus [Liskov Scheifler 1983 Actions] belong in this category. CSP [Hoare 1978 CACM Communicating] also belongs to this group, although it is less clearly based on an abstract data type model. In CSP, individual processes encapsulate data. Other processes may access the encapsulated data only by an exchange of messages. The process owning the data resource defines all the operations on the data and localizes data access. Synchronization is not as well localized, because the synchronization depends on the "matching" of input and output commands distributed among many processes.

Although there are many synchronization constructs that support resource modularity, relatively few of them permit real concurrency within the synchronization module. Such constructs do not allow encapsulated concurrency. For instance, monitors allow at most one process to be active at a time. In order to allow multiple processes to access a resource simultaneously (as for reader processes in the well-known readers and writers problem [Courtois]), a monitor is used only to implement a pre-read/ post-read and pre-write/ post-write protocol, which is called before and after a call to an external *read* or *write* routine [Hoare 1974 CACM Monitors]. There is no assurance that the protocol will be followed. Deadlock or data corruption may result if it is not. The lack of encapsulated concurrency also makes it difficult to nest modules or to otherwise structure concurrency. Structured concurrency is needed to develop atomic action and fault-tolerant systems [Campbell Randell, Jalote Campbell recoverability, Liskov Scheifler actions, Best Randell]. Concurrent Pascal (which is monitor based) [Brinch_Hansen 1975], DP [Brinch_Hansen 1978], Ada [Defense 1981] and CSP [Hoare 1978 CACM Communicating] all fail to encapsulate concurrency. Argus [Liskov Scheifler actions, Liskov Weihl], provides encapsulated concurrency, but with severe restrictions to ensure recoverability. In effect, the restrictions eliminate most parallelism. SR [Andrews TOPLAS 1981], Path Pascal (PP) [Campbell Kolstad SIGPLAN 1980 overview] Distributed Path Pascal (DPP) [Kolstad 1983 Distributed], serializers [Hewitt Atkinson 1979 TOSE], and MCP [Bahsoun] do allow specification of encapsulated concurrency.

Synchronization modularity refers to the ability to specify synchronization and scheduling constraints separate from the specification of the resource data abstraction. This additional structuring device aids in system development, but also benefits the validation of design and code. Modular synchronization may also make it possible to develop libraries of synchronizers and schedulers. The isolation of timing aspects contributes to real-time programming as well.

Few constructs provide synchronization modularity. Among those are Path Pascal (PP and DPP) [Campbell Kolstad 1980 SIGPLAN overview], sentinel processes [Ramamritham Keller 1981], and serializers [Hewitt Atkinson 1979]. Serializers are implemented in a LISP environment. Sentinel processes appear to be the imperative language analog. Both combine built-in counters with queueing primitives to allow modular specification of synchronization. These constructs appear to be well suited to FIFO scheduling problems and variants of the reader/writer problem, but less flexible than desired [Bloom 1979 Evaluating]. Path Pascal encapsulates most synchronization specifications in a path expression. This often provides a high degree of synchronization modularity. The synchronization modularity is lost when conditional synchronization or scheduling is specified. These must be programmed using nested objects. This results in loss of modularity as well as inefficiency due to the implicit scheduling applied at each level of nesting [Grass 1984 Mediators, Campbell Donnelly 1984]. In order to maintain synchronization modularity, synchronization data must be encapsulated. In addition, there must be support for conditional synchronization and scheduling.

For practical embedded distributed systems, it is important not to overly constrain the system implementer in terms of possible synchronization and scheduling. Synchronization schemes that enforce atomic recoverable transactions (such as Argus and Clouds [Allchin McKendry 1983 Recovery , Allchin Mckendry 1983 Support]) are essentially ruled out, as they severely limit what can be specified.

Other schemes allow more flexibility in what can be specified, but make the expression of some kinds of constraints difficult. As we noted above, Sentinel Processes make FIFO scheduling problems quite easy to specify, but specification of operation sequences is complicated [Ramamritham Keller 1983 TOSE]. In Path Pascal it is easy to specify sequences of operations, but implementing scheduling or conditional synchronization is complicated. It should be possible to express constraints in terms of resource history, resource and synchronization state and information about pending requests.

The configuration of concurrent systems raises other questions about flexibility. Many proposed language constructs for writing distributed systems rely on static systems. In DP and Concurrent Pascal [Brinch Hansen 1975, Brinch Hansen 1978] processes and modules are instantiated at system creation and never terminate. This is not reasonable for real systems that sometimes require on-the-fly reconfiguration, partial reinitialization, or simply need to print a diagnostic message before failing. DP and Concurrent Pascal do not support resource nesting, so all resources are alive for the duration of the system, which is essentially infinite.

Other constructs allow processes and objects to come and go, but are inflexible in other ways. Frequently communication paths are static. CSP is an extreme case of this [Hoare 1978 CACM Communicating] in which the sender and the receiver of a message need to know each other's name. This feature of CSP makes it impossible to write libraries of services. CSP was meant to be an exercise in input/output rather than a complete language proposal. Some CSP successors, such as OCCAM [May Occam April SIGPLAN], have attacked this problem by introducing ports. SR [Andrews 1982 Mechanisms] has a similar communication problem. Server processes and clients

are tied in a one-to-one relationship that is explicit and rigid.

Most synchronization proposals allow servers to honor requests fr .n anonymous clients. This is a flexible arrangement, but occasionally there are cases in which the client's identity must be known. Some language constructs provide this information (PLITS [Feldman 1979 High]), but more often it is left up to the implementer. The *mediator* proposal supports dynamic creation and termination of mediators and flexible communication paths. It also provides a means of identifying clients.

7.3. A Proposal

The *mediator* combines several proposals in an attempt to provide a solution to the problems that are outlined above.

1) The Path Pascal object [Campbell Kolstad 1980 Overview] serves as the model for the mediated object. The path expression is replaced by the mediator. The external view of the object should change little.

2) The Path Pascal object initiation block is replaced by initiation and termination code encapsulated in the mediator.

3) The essential control structure within the mediator is an adaptation of Dijkstra's guarded commands [Dijkstra Guarded]. Our adaptation uses *delay semantics* [Brinch_Hansen Distributed Processes] rather than Dijkstra's *abort* semantics.

4) Requests are associated with unique *keys* that allow the mediator to manipulate requests and implement scheduling.

5) Guards may contain *status tests* to inquire about pending requests, and boolean tests which may refer to data contained in pending requests [Hoare 1978 CACM, Feldman 1979 High].

6) The mediator controls request execution by commands allowing coupled and uncoupled client process execution [Ramamritham Keller 1983 TOSE]. There is an explicit command to return results to a client.

7) *Parallel guards* are used to multi-program the mediator. Mediator execution is guaranteed atomic between guard evaluations.

Section two explains these features in greater detail, presenting a syntax and examples.

Section 3 discusses the problems of specifying a new language construct. This paper relies on a BNF grammar to specify syntax, and informal descriptions of the semantics supported by examples. Ultimately, a more formal description must be produced. A meta-language description supported by temporal logic axioms will be used.

Section 4 examines implementation aspects. Many of the individual components of the mediator have been implemented in other languages. The main difficulty is combining these in an efficient manner.

The proposal presented here is preliminary. Some features of the syntax and semantics may change as a formal description is developed, and as implementation issues become more central.

APPENDIX F

Path Pascal Compiler Documentation

The original user manual[1] for Path Pascal mentions the existence of interrupt processes" but does not give the details of their implementation. As a feasibility study, a temporary implementation of interrupt processes has been made for a version of Path Pascal operating on Berkeley version 4.2BSD Unix.

The central feature of interrupt processes under Path Pascal is the *doio* system call. Its arguments are implementation-defined, and it performs the functions of (possibly) initiating an I/O transfer, and then blocking the currently active process until an interrupt occurs. When the interrupt does occur, the scheduler returns both the interrupt process and the process that was interrupted to the ready queue, and then dispatches them according to their priorities. In this way, interrupt-driven pre-emptive scheduling can be implemented by giving the process that issues the *doio* request priority over normal processes.

In the temporary Unix implementation, doio was implemented to take a single argument: the Unix signal number which the process wishes to await. The occurrence of the corresponding event causes the interrupt and context switch. In testing this mechanism, two signals were chosen for use: SIGINT, which reports that the user has issued an interrupt request using a control sequence at the terminal; and SIGALRM, which reports that a program-controlled interval timer has expired.

The existence of a *doio* request capable of handling these signals, together with support functions to change a process's priority and to set the real-time-clock, provided enough power to implement a reasonably complex demonstration program. The program that was chosen was the real-time scheduler" described in Chapter 7 of Per Brinch Hansen's *Architecture of Concurrent Programs*.

This system, inspired by an earlier program designed for process control at an ammonia nitrate plant, allows a fixed number of concurrent tasks to be carried out periodically with frequencies chosen by the operator. The functions available to the operator are to tell the system what time it is, say when a task should first be executed, and say how often a task must be repeated. Each task, the program that manages the real-time clock, and the program that interacts with the operator, are all modelled as concurrent processes in the implementation.

The conversion of this system from the original Concurrent Pascal to Path Pascal went quite smoothly; only a few minor problems were noted. The ease of the conversion has convinced the author that Concurrent Pascal has at least the expressive power of Path Pascal. The only significant changes (other than minor differences in syntax) required between the two implementations related to the fact that Concurrent Pascal supports only dynamically created and initialized objects, whereas Path Pascal allows objects to be created statically, and initialized before the main program is entered. The static nature of objects means that their initialization procedures may not accept parameters; some of the data supplied to the monitors at initialization time in the Concurrent Pascal implementation had to be passed as parameters to entry procedures in the Path Pascal version.

This author, in doing the conversion, copied many of Brinch Hansen's objectionable programming practises. In particular, the system is dependent throughout on knowing in advance the identifying numbers assigned to each process by the runtime system. This assumption is fraught with peril (and, in fact, caused this author some debugging problems, since the process indices changed in the Path Pascal implementation from their original values in Concurrent Pascal). This problem was the only serious impediment when the program was being debugged; all of Brinch Hansen's test harnesses were also converted to Path Pascal and used when debugging the system. None of his test cases required more than four runs before giving correct results; most executed correctly on the first attempt.

---

[1] Grunwald, Dirk C., *Path Pascal User Manual*, University of Illinois at Urbana-Champaign, Urbana, Illinois, May 1985.

[2] Brinch Hansen, Per, *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

The reader who is interested in the details of the implementation is invited to compare the accompanying source code listing with Brinch Hansen's original program.

```
type bellkey = object
path (signal; await) end;

const SICINT = 2;

entry procedure signal; begin end;
entry procedure await; begin end;

process interruptor;
begin
        RTS_setprio (7);
        while (true) do begin
                doio (SICINT);
{               delay (1);              }
                signal;
        end;
end;

initially;
begin
{       rtsdebugging(3);                }
        interruptor;
end;

end {bellkey};
```



```
#include "types.i"

type clock = object
path 1: (value, correct, tick) end;

var seconds: real;

entry function value: real;
begin
        value := seconds;
end;

entry procedure correct (time: real);
begin
        seconds := time;
end;

entry procedure tick;
begin
        seconds := seconds + 1.0;
        if (seconds >= oneday) then
                seconds := seconds - oneday;
end;

initially;
begin
        seconds := 0.0;
end;

end {clock};
```

```
const SIGALRM = 14;

process clockprocess (var watch: clock; var schedule: timetable;
                      var waiting: taskqueue);
begin
        setclock (1, 0, 1, 0);
        while (true) do begin
                doio (SIGALRM);
                watch.tick;
                schedule.examine (watch.value, waiting);
        end;
end {clockprocess};
```



```
type fifo = object
        path setsize end;
        path arrival end;
        path departure end;
        path empty end;
        path full end;

var head: integer;
    tail: integer;
    length: integer;
    limit: integer;

entry procedure setsize (l: integer);
begin
        limit := l;
end;

entry function arrival: integer;
begin
        arrival := tail;
        tail := tail mod limit + 1;
        length := length + 1;
end;

entry function departure: integer;
begin
        departure := head;
        head := head mod limit + 1;
        length := length - 1;
end;

entry function empty: boolean;
begin
        empty := (length = 0);
end;

entry function full: boolean;
begin
        full := (length = limit);
end;

initially;
begin
        head := 1;
        tail := 1;
        length := 0;
        limit := 1;
end;

end {fifo};
```

```
process operatorprocess (var typeuse: resource; var tasklist: taskset;
                         var watch: clock; var schedule: timetable);

var operator: terminal;
    bell: bellkey;
    letters, digits: set of char;
    ok: boolean,
    ch: char;
    command: identifier;

procedure help;
begin
        if ok then begin
                operator.writetext ('\#');
                operator.writetext ('Try again:\#');
                operator.writetext ('   start(task,hour:min:sec)\#');
                operator.writetext ('   period(task,hour:min:sec)\#');
                operator.writetext ('   stop(task)\#');
                operator.writetext ('   time(hour:min:sec)\#');
                ok := false;
        end;
end;

procedure nextchar;
begin
        if ok then
                repeat ch := operator.read until ch <> ' ';
end;

procedure skipchar (delim: char);
begin
        if ch = delim then nextchar
                      else help;
end;

procedure readint (var int: integer);
const maxint = 32767;
var digit: integer;
begin
        int := 0;
        if not (ch in digits) then help
        else
         while (ch in digits) & ok do begin
                digit := ord (ch) - ord ('0');
                if int > (maxint - digit) div 10 then help
                                                 else int := 10 * int + digit;
                nextchar;
        end;
end;

procedure readid (var id: identifier);
var length: integer;
begin
        id := '               ';
        if not (ch in letters) then help
        else begin
```



```
                length := 0;
                while ((ch in letters) or (ch in digits))
                    & (length < idlength) do begin
                        length := length + 1;
                        id [length] := ch;
                        nextchar;
                end;
        end;
end;

procedure readtime (var time: real);
var hour, min, sec: integer;
begin
        readint (hour); skipchar (':');
        readint (min); skipchar (':');
        readint (sec);
        if (hour > 23) or (min > 59) or (sec > 59) then help;
        if ok then time := onehour * hour + onemin * min + sec;
end;

procedure start;
var id: identifier;
    time: real;
begin
        skipchar ('('); readid (id);
        skipchar (','); readtime (time);
        skipchar (')');
        if ok then
                if tasklist.member (id) then
                        schedule.start (tasklist.task (id), time)
                else
                        operator.writetext ('     Task unknown.\#');
end;

procedure period;
var id: identifier;
    time: real;
begin
        skipchar ('('); readid (id);
        skipchar (','); readtime (time);
        skipchar (')');
        if ok then
                if tasklist.member (id) then
                        schedule.period (tasklist.task (id), time)
                else
                        operator.writetext ('     Task unknown.\#');
end;

procedure stop;
var id: identifier;
begin
        skipchar ('('); readid (id);
        skipchar (')');
        if ok then
                if tasklist.member (id)
                        then schedule.stop (tasklist.task (id))
```

```
                        else operator.writetext ('     Task unknown.\#');
end;

procedure correct;
var time: real;
begin
        skipchar ('('); readtime (time);
        skipchar (')');
        if ok then watch.correct (time);
end;

begin {operator process}
        letters := ['a' .. 'z'];
        digits := ['0' .. '9'];
        while (true) do begin
                bell.await;
                typeuse.request;
                ok := true;
                operator.write (chr (7) {bell});
                operator.writetext ('Type command:\#');
                nextchar;
                readid (command);
                if command = 'start       ' then start else
                if command = 'period      ' then period else
                if command = 'stop        ' then stop else
                if command = 'time        ' then correct
                                                 else help;
                operator.writetext ('OK.\#');
                typeuse.release;
        end;
end;
```



```
type QUEUE = object
        path (join; continue; depart) end;
        path 1: (join, continue, depart) end;
        path delay end;

entry procedure join; begin end;
entry procedure continue; begin end;
entry procedure depart; begin end;
entry procedure delay;
begin
        join;
        depart;
end;

end {QUEUE};
```

Jun 19 15:11 1985 resource.p Page 1

```
type resource = object
path 1: (request; release) end;
entry procedure request; begin end;
entry procedure release; begin end;
end;
```

Jun 22 02:20 1985 system.p Page 1

```
program system (input, output);

#include "procs.i"
#include "types.i"
#include "resource.i"
#include "taskqueue.i"
#include "taskset.i"
#include "clock.i"
#include "timetable.i"
#include "bellkey.i"
#include "terminal.i"
#include "taskprocess.i"
#include "clockprocess.i"
#include "operatorprocess.i"

var
        typeuse: resource;
        waiting: taskqueue;
        tasklist: taskset;
        watch: clock;
        schedule: timetable;

begin
        tasklist.include ('scan        ', 2);
        taskprocess (typeuse, waiting, tasklist, watch);
        tasklist.include ('flow        ', 3);
        taskprocess (typeuse, waiting, tasklist, watch);
        tasklist.include ('log         ', 4);
        taskprocess (typeuse, waiting, tasklist, watch);
        clockprocess (watch, schedule, waiting);
        operatorprocess (typeuse, tasklist, watch, schedule);
end.
```

Jun 22 01:58 1985 taskprocess.i Page 1

```
process taskprocess (var typeuse: resource; var waiting: taskqueue;
                var tasklist: taskset; var watch: clock);

var operator: terminal;
    id: identifier;

procedure writeid (id: identifier);
var i: integer;
begin
        for i := 1 to 24 * (tasklist.task (id) - 2) do operator.write (' ');
        for i := 1 to idlength do operator.write (id [i]);
        operator.write (' ');
        operator.write (chr (7) {bell});
end;

procedure writetime (time: real);
var hour, min, sec: integer;
    rem: real;
begin
        hour := trunc (time / onehour);
        rem := time - hour * onehour;
        min := trunc (rem / onemin);
        sec := trunc (rem - min * onemin);
        operator.writeint (hour);
        operator.write (':');
        operator.writeint (min);
        operator.write (':');
        operator.writeint (sec);
        operator.writetext ('\@');
end;

begin
        tasklist.me (id);
        while (true) do begin
                waiting.preempt;
                typeuse.request;
                writeid (id);
                writetime (watch.value);
                typeuse.release;
        end;
end;
```

Jun 19 16:00 1985 taskqueue.p Page 1

```
#include "types.i"
#include "procs.i"

type taskqueue = object
        path preempt end;
        path resume end;

var waiting: processqueue;

entry procedure preempt;
begin
        waiting [RTS_getpid] . delay;
end;

entry procedure resume (task: processindex);
begin
        waiting [task] . continue;
end;

end;
```

Jun 21 22:01 1985 taskset.p Page 1

```
#include "types.i"
#include "procs.i"

type taskset = object
        path 1: (include, member, task, me) end;

var table: array [processindex] of identifier;

function index (id: identifier): processindex;
var i, j: processindex;
begin
        i := 1;   j := processcount;
        while (i < j) do begin
                if (table [i] = id) then j := i
                                    else i := i + 1;
        end;
        index := i;
end;

entry procedure include (id: identifier; task: processindex);
begin
        table [task] := id;
end;

entry function member (id: identifier): boolean;
begin
        member := (table [index (id)] = id);
end;

entry function task (id: identifier): processindex;
begin
        task := index (id);
end;

entry procedure me (var id: identifier);
begin
        id := table [RTS_getpid];
end;

initially;
var task: processindex;
begin
        for task := 1 to processcount do
                table [task] := '            ';
end;

end {taskset};
```

Jun 19 14:59 1985 terminal.p Page 1

```
#include "types.i"
#include "typewriter.i"

type terminal = object
        path write end;
        path writetext end;
        path writeint end;
        path read end;

var device: typewriter;

entry procedure write (c: char);
begin
        device.xwrite (c);
end;

entry procedure writetext (text: line);
var i: integer;
    c: char;
begin
        i := 1; c := text [1];
        while (c <> '@') do
        begin
                if (c = '\') then c := chr (10);
                        { Can't get newlines into string }
                device.xwrite (c);
                i := i + 1;
                c := text [i];
        end;
end;

entry procedure writeint (i: integer);
var digits: array [1 .. 13] of char;
    rem: integer;
    length: integer;
begin
        rem := i; length := 0;
        repeat begin
                length := length + 1;
                digits [length] := chr (rem mod 10 + ord ('0'));
                rem := rem div 10;
        end until rem = 0;
        for length := length downto 1 do
                device.xwrite (digits [length]);
end;

entry function read : char;
begin
        read := device.xread;
end;

end {terminal};
```

```
#include "types.i"
#include "procs.i"
#include "taskqueue.i"

type timetable = object
        path 1: (start, period, stop, examine) end;

type taskschedule = record
        active: boolean;
        start: real;
        period: real
end {taskschedule};

var table: array [processindex] of taskschedule;

function reached (time, start: real): boolean;
var diff: real;
begin
        diff := time - start;
        if (abs (diff) >= halfday)
                then reached := (diff < 0.0)
                else reached := (diff >= 0.0);
end;

entry procedure start (task: processindex; time: real);
begin
        table [task].active := true;
        table [task].start := time;
end;

entry procedure period (task: processindex; time: real);
begin
        table [task] .period := time;
end;

entry procedure stop (task: processindex);
begin
        table [task] .active := false;
end;

entry procedure examine (time: real; var waiting: taskqueue);
var task: processindex;
begin
        for task := 1 to processcount do begin
            with table [task] do
                if (active) then begin
                        if (reached (time, start)) then begin
                                waiting.resume (task);
                                start := start + period;
                                if start >= oneday then
                                        start := start - oneday;
                        end;
                end;
        end;
end;
```



```
initially;
var task: processindex;
begin
        for task := 1 to processcount do
                table [task] .active := false;
end;

end {timetable};
```

Jun 21 22:11 1985 types.i Page 1

```
#include "queue.i"
const linelength = 72;
      processcount = 20;
      idlength = 12;
      onemin = 60.0 {seconds};
      onehour = 3600.0 {seconds};
      halfday = 43200.0 {seconds};
      onsday = 86400.0 {seconds};

type line = array [1..72] of char;
     processqueue = array [1 .. 20] of QUEUE;
     processindex = 1 .. 20;
     identifier = array [1 .. idlength] of char;
```

Jun 19 14:59 1985 typewriter.p Page 1

```
#include "procs.i"

procedure putchar (c: integer); external;
function getchar : integer; external;

type typewriter = object
        path xwrite end;
        path xread end;

entry procedure xwrite (c: char);
begin
        putchar (ord (c));
end;

entry function xread : char;
begin
        xread := chr (getchar);
end;

end {typewriter};
```

APPENDIX G

Path Pascal Compiler Distribution List

Partial Path Pascal VAX Compiler Distribution List.

1. Tungning Cherng Digital Equipment Corp. Continental Boulevard, MK02-1/H10 Merrimack, NH 03054
2. Arnold Robbins School of Information and Computer Science Georgia Inst. of Technology 225 North Avenue N.W. Atlanta, Georgia 30332
3. Joan Eslinger Gould Computer Systems 1101 E. University Urbana, Il 61801
4. Ray Ford, University of Iowa, Department of Computer Science, Iowa City, Iowa 52242
5. Lawrence Fitzpatrick National Library of Medicine Lister Hall Center 8600 Rockville Pike Bethesda, MD 20209 Bldg 38A, Room 8N815
6. Mike Finn Nuclear Physics Lab University of Illinois, Urbana Illinois
7. Ginnie Lo, Dept. of Computer & Information Science, 64 Prince Lucien Campbell Hall University of Oregon, Eugene, OR 97403
8. Professor K. H. Kim, University of South Florida, College of Engineering, Dept. of Computer Science and Engineering, Tampa Florida, 33620
9. Eric Kaylor Engineering Computer Laboratory Washington University Campus Box 1045 St Louis, Mo. 63130

APPENDIX H

Path Pascal Program Examples

PATH PASCAL USER MANUAL

Original Manual by:
Robert B. Kolstad
Roy H. Campbell

Revised for PPC:
Dirk C. Grunwald

Department of Computer Science

University of Illinois at Champaign-Urbana

Urbana, Illinois 61801

217-333-0215

ABSTRACT

Original Manual: 1/29/80
Revised: 11/1/84
Printed: 8/9/85

(Funded in part by NASA Grant NSG1471 and NSF Grant MCS 77-09128)

1. Introduction

Path Pascal was designed to investigate the benefits and problems that arise when Path Expressions are combined with a language to provide a system programming tool. Instead of altering the Pascal language extensively, a minimal number of features was added such that Pascal programs still compile and execute. The language can be used for instruction or construction of example system programs. It has also proven useful for discrete systems simulation and performance analysis.

Path Expressions were introduced as a technique for specifying process synchronization by [Campbell & Habermann, 74], and further discussed by [Habermann, 75], [Lauer & Campbell, 75], [Flon & Habermann, 76], [Andler, 79] and [Campbell, 77]. Variations of the Path Expression idea have been proposed by [ONERA CERT, 77] and notations that are similar to paths that model system behavior have been developed independently by [Shaw, 77] and [Riddle, 76]. A specification language has also been designed [Lauer & Shields, 78] based upon the use of a Path Expression notation.

The current system, called 'ppc', is an extension of the Berkley Pascal compiler [Joy, Graham & Haley, 80], and is largely intended to build upon the work started with the Experimental Path Pascal compiler which was based on the P4 subset of Pascal [Kolstad & Campbell, 79]. Pascal was augmented with an encapsulation mechanism (see chapter 2), Open Path Expressions [Campbell, 77] (see chapter 3), and a process mechanism (see chapter 4). Open Paths are integrated with the encapsulation mechanism to enforce a strict discipline upon the programmer to describe shared data objects. All access to encapsulated data is performed by operations synchronized by Open Paths. A process invoking such operations may execute the operation only if

permitted by the Open Path Expressions associated with the shared data object.

The following chapters describe Path Pascal in more detail. Motivations for the design of Path Pascal are discussed further in [Miller, 78], [Campbell & Kolstad, 79a], [Campbell & Kolstad, 79b], [Campbell & Kolstad, 80], [Horton & Campbell, 80], [Kolstad & Campbell, 80], and [McKendry & Campbell, 80]. A description of Pascal can be found in the Pascal Report [Jensen & Wirth, 75], with the Berkley Pascal extensions noted in [Joy, Graham & Haley, 80]. The additional Path Pascal syntax and changes to the Berkley Pascal compiler are listed in Appendix A. The differences between PPC and Experimental Path Pascal are listed in Appendix B. The algorithim to generate semaphore P and V operations from Open Path Expressions is given in Appendix C. Appendix D contains several sample programs. Appendix E contains information useful for debugging PPC programs.

2. Data Encapsulation

2.1. Introduction to Objects

Encapsulating data and definitions of operations on that data ensures that only intended accesses and transformations are made to an information structure. The addition of a synchronisation mechanism to data encapsulation allows protection from asynchronous access. In Path Pascal, an encapsulation mechanism called an *object* specifies access, transformation, and synchronization. An object's data and code are accessible to other parts of the Pascal program only by explicit declaration of entry operations. Objects are implemented as a restricted extension of the Pascal structured *type* facility.

2.2. Object Declaration

Each object begins with the declarator *object,* then specifies the synchronization for the object via one or more Path Expression (see chapter 3), followed by the Pascal declarations (consts, types, vars), the routines of the object (procedures, functions, processes, initialization and finalization procedures) in appropriate order for scope consideration, and finally an *end* token. The const, type, var, and routine specifications are expressed as in standard Pascal and have the same actions.

The object defines a block which follows the scope rules of standard Pascal; though exported procedures, functions, and processes have the additional attribute of appearing as defined in the scope containing the object. Only exported procedures, processes, and functions are available to enclosing scopes for examination and manipulation of encapsulated data.

Object types may be declared with explicit names in a *type* statement or implicitly (along with instaniation) using the *var* statement. Object names defined as types may be used to declare any number of object instaniations in *var* statements. Once instaniated, each object has its own copies of storage, the object's operations, and synchronization information.

Objects may be nested within structures or within other objects. Recursive object instaniations, like recursive record instaniations, are flagged as errors during compilation.

Pointers to objects are declared in *var* or *type* statements similar to declarations of pointers to other data types. The body of the referenced object must be processed before any qualifications on the pointer can be made in other object definitions. Dynamic instaniations may be created by executing the standard procedure named *new* with a pointer argument. Pointers to objects permit the construction of encapsulated and recursive data structures.

2.3. Operations

Functions, processes, and procedures whose names are exported from an object are known as *operations.* They are differentiated from internal procedures, processes, and functions by prefixing their declaration by the token *entry.* Operations, like all routines within an object, can invoke other operations and routines within the object (as long as scope considerations are satisfied). Synchronization is applied as usual for invoked operations.

Operations within an object are invoked as standard procedures. Outside the object, however, the name of the object's instaniation (or a dereferenced pointer to the object's instaniation) and a period must precede the name of the operation to be invoked. Operations may be invoked recursively, even though a deadlock might eventually result.

2.4. Exported Types

Exported types were defined in Experimental Path Pascal, but have not yet been implemented in PPC. Exported types can be simulated through the judicious use of *#include* files.

2.5. Path Declaration

The object's Path Expression specifies the synchronization constraints of the object's operations. The syntax for Path Expressions has changed slightly from Experimental Path Pascal, and these changes are mentioned in Chapter 3, which discusses Path Expressions in detail.

2.6. Initialization Procedure

The initialization procedure of an object is an optional procedure which is executed upon instaniation of the object. The procedure is a full Pascal procedure with declarations and subprocedures allowed. The syntax is different from a standard procedure as no parameters are allowed. Standard scope rules apply. The initialization procedure must appear after all the procedures within an object and before the finalization procedure

An initialization block is composed of the token *initially* followed by a semicolon, declarations and a procedure body. Any objects declared within the object are initialized and available from within the initialization procedure. However, references to objects defined in the same scope as the current object are to be avoided since the order of object initialization is undefined.

2.7. Finialization Procedure

The finalization procedure matches the purpose of the initialization procedure. It is executed immediately before the resources for the object are deallocated. The finalization procedure must be the last procedure in the object.

A finalization procedure is composed of the token *finally* followed by a semicolon, declarations and a procedure body. Any objects declared within the object are disposed of after the finalization procedure is executed. As with initialization procedures, references to objects declared in the same scope as the current object are to be avoided as the order of finalization procedure execution is undefined.

2.8. Externally Declared Objects

It is possible to compile an object implementation and link it into the final load image using externally declared objects. To access the entry points within the object, an object outline must be provided. The syntax for this follows the Berkley Pascal model for separate compilation.

External objects can only be declared at the global level, and then only as a type, as this provides a one-to-one naming for the object and the code body associated with the object. Within the object, only the entry procedures, functions and processes are listed. The linking phase of the compiler ensures that no type violations have occurred. An example is given at the end of this section.

2.9. Implementation Details

Assignments between variables containing objects are not permitted since they are semantically unsound. An object maintains not only the data contained within the object but also the state information of processes which are blocked waiting to access that information. Object variables or structured variables containing objects must always be passed as reference parameters to routines. Objects are actually allocated from the global heap, and are disposed of when a procedure invocation ends. Thus, the size of an object, for purposes of computing the size of an

activation record for processes, is the same as the size of a pointer.

2.10. Examples of Objects

The example below shows the declaration of a typical *object* type, its instaniation, and two invocations:

```
const
    nbuf = 5;

type
    bufrange = 1..nbuf;
    ring = object

        path nbuf:(1:(put);1:(get)) end;

        var
            buffer: array[bufrange] of char;
            inp, outp:  bufrange;

        entry procedure put(x: char);
        begin
            inp := (inp mod nbuf) + 1;
            buffer[inp] := x
        end;

        entry function get: char;
        begin
            outp := (outp mod nbuf) + 1;
            get := buffer[outp]
        end;

        initially;
        begin
            inp := nbuf;
            outp := nbuf
        end;
    end;

var
    buf: ring;
    c: char;

begin
    buf.put('a');
    c := buf.get
end.
```

The initialization block sets the pointers to appropriate values for standard ring buffering. The operation 'put' is called to deposit characters within the buffer, 'get' retrieves them. The Path Expression eliminates the need for any further synchronization specification of the head and tail of

the buffer.

2.11. Syntax for Objects

Backus Naur Form for each of the new specifications is shown below:

```
obj_type ::= object
        <path_decl_part>
        { <object-declerations> }*
        { <initially> }
        { <finally> }
        end

object-declerations ::=
        { <pascal-declerations> } |
        <entry-header> <body>

<entry-header> ::=
        entry procedure <id> ( <parameters> ); |
        entry function <id> ( <parameters> ) : type ; |
        entry process <id> { [ constant ] } ( parameters );

<initially> ::=
        initially; <body>

<finally> ::=
        finially; <body>
```

2.12. Examples of External Objects

The following example illustrates the use of an externally compiled object. This requires three files to be processed. The first contains the actual definition for the object:

bufferobj.p:

```
const
    nbuf = 5;

type
    bufrange = 1..nbuf;
    ring = object
        path nbuf:( 1:(put); 1:(get) ) end;

    < .. The rest of the type 'ring' is as in the previous example .. >
```

bufferobj.i:

```
const
    nbuf = 5;

type
    bufrange = 1..nbuf;
    ring = external object
        entry procedure put( c : char );
        entry function get : char;
        end;
```

usering.p:

```
program use(output);

#include "bufferobj.i"

var
    buf : ring;
    c : char;
begin
    buf.put('a');
    c := buf.get;
end.
```

The commands given to the UNIX shell to compile these modules and link them together would be the following:

```
% ppc -c bufferobj.p
% ppc -c usering.p
% ppc -o usering usering.o bufferobj.o
```

2.13. Syntax for External Objects

The syntax is similar to the object:

```
<external-object> ::= external object
        <entry-header-list>
        end;

<entry-header-list> ::=
        <entry-header> |
        <entry-header-list> <entry-header>
```

The major difference is that the only entry definitions may appear. The previous restrictions (must appear at the global level, and can only be used in type declarations) also hold, but have not been shown, although they do hold in the actual language grammar. An additional restriction is that the size of processes may not be specified in an <external-object>.

3. Path Expressions

3.1. Introduction to Path Expressions

An *Open Path Expression* specifies the synchronization constraints for a possibly concurrent set of process, procedure, and function executions within objects. This static description allows code to be written without any explicit reference to synchronization primitives. In Experimental Path Pascal, each object could contain a single Path Expression with multiple references to object entry points. In PPC, each object can contain multiple Path Expressions with at most one reference to any entry point within a single Path Expression.

Path Expressions specify the sequential and concurrent synchronization for the object. Since only the entry operations can be accessed from outside the object, the information structure can be protected from unsafe sequences.

Normally, the order of invocation of procedures is unknown until the invocation occurs since processes can execute asynchronously. Path Expressions allow three distinct kinds of constraints to be specified: sequencing (denoted by ';'), resource restriction (denoted by 'n:( )'), and resource derestriction (denoted by '[ ]'). Each of these can be combined with the other forms to provide complex synchronization constraints and several constraints can be contained in a single Path

Expression. These forms are described with examples below.

A Path with no synchronization information consists of a comma separated list of operation names surrounded by path and end. The Path below:

path name1, name2, name3 end

imposes no restriction on the order of invocation of the operations and no restriction on the number of concurrent executions of 'name1', 'name2', and 'name3'.

The sequencing mechanism imposes an order on procedure executions. The order is specified by a semi-colon separated list. In the example below:

path first; second; third end

one execution of operation 'first' must complete before each execution of 'second' may begin, and one execution of 'second' must complete before each execution of 'third' can begin. Of course, the execution of a 'third' or 'second' in no way inhibits the initiation of 'first'; several operations may be executing concurrently.

Limited resources (e.g., line printers) occasionally make it desirable to limit the number of concurrent executions of an operation. The resource restriction specification allows concurrent execution of operations to proceed until the restriction limit is reached. Restrictions are denoted by surrounding the expression to be restricted by parentheses and preceding it with the integer restriction limit and a colon. The restriction below:

path 2:(ttyhandler) end

allows only two invocations of 'ttyhandler' to proceed concurrently. Any invocation of 'ttyhandler' will wait until less than two executions are active before it begins execution. The number preceding the colon in a restricter can be thought of as the number of resources for which the operation competes. A critical section, in which only a single resource is to be shared, is easily specified. In the example below:

path 1:(routine1, routine2, routine3) end

only one of the three operations can be active at a time. Restricters may be positive integers or positive constants.

For some applications it is convenient to process all calls to an operation once that operation's execution has begun. Such a situation might occur when a large spooler is brought into memory to process I/O requests. The specifier denoting 'derestriction' of a list of operations is shown by surrounding the list in square brackets. The path below:

path setup; [spooler] end

requires 'setup' to be executed before each sequence of calls to 'spooler', but once 'spooler' has begun execution, its invocations proceed to execution until all executions have terminated. Afterwards, 'setup' must again complete before any 'spooler' can proceed.

Each of the forms above (without **path** and **end**) can be considered to be a subexpression of a Path. Subexpressions may be combined (with the optional use of parentheses for clarity) in the formats above to yield complex paths, with the restriction that an entry name may appear only once within a single path expression. Normally, the sequencing operator (";") has higher precedence than the alternation operator (",").

It is possible to specify multiple Path Expressions, one right after the other. The restrictions

described by the Path Expressions are imposed in the order listed. Thus, if we wish to specify a resource restriction and a sequence operation, we could use:

```
path reader;writer end
path 1:(reader, writer, status) end
```

Note that procedure 'status' appears only in the last Path Expression. If we had reversed the order of the two Path Expressions:

```
path 1:(reader, writer, status) end.
path reader;writer end
```

the system would quickly deadlock if a writer were to access the object before a reader, since the writer would pass the restriction path, but would block on the sequencing path.

The ability to have multiple Path Expressions is a change from Experimental Path Pascal, where the same semantics could be expressed using multiple references to object entries within a single path expression. The syntax has been changed in an effort to clarify the order of evaluation of the Path Expressions. This syntax allows a less expressive notation, however it can be conjectured that examples which can not be easily expressed in the current syntax are most likely inherently unclear to begin with.

3.2. Examples of Open Paths

1. **path a end ;**

   Routine 'a' can execute at any time, and any number of 'a's can execute concurrently. No synchronization is specified.

2. **path a, b, c end ;**

   Routines 'a', 'b', and 'c' can execute at any time. Any number of each one can execute concurrently. No synchronization is specified.

3. **path a; b end ;**

   Routine 'a' can be executed at any time, but 'b' can only begin if the number of 'b's that have begun execution is less than the number of 'a's that have completed.

4. **path 1:(a) end ;**

   Routine 'a' must be executed sequentially (only one 'a' active at a time).

5. **path 2:(a) end ;**

   At most two executions of routine 'a' may proceed concurrently.

6. **path 1:(a), b end ;**

   Multiple invocations of routine 'a' proceed in sequential execution. No restriction is placed on routine 'b'.

7. **path 1:(a); 1:(b) end ;**

   Both 'a' and 'b' are critical sections. A maximum of one each of 'a' and one 'b' can execute concurrently.

8.

```
path 6:(5:(a), 4:(b)) end ;
        or
path 5:(a), 4:(b) end ;
path 6:(a, b) end ;
```

As many as five invocations of 'a' and four of 'b' can proceed concurrently as long as the limit of six total executions is not exceeded.

NOTE: This is not equivalent to

```
path 6:(a, b) end ;
path 5:(a), 4:(b) end ;
```

since this version is different in that it if six calls to 'b' were past the first restriction, and four executions of 'b' were underway, they would block any further calls to 'a' from proceeding since the first restricter is full. This path provides less concurrency than the previous paths.

9.

```
path 5:(a;b) end ;
        or
path a;b end ;
path 5:(a) end ;
```

No more than five executions of routine 'a' and routine 'b' can be proceeding concurrently. Each execution of 'b' must be preceded by an execution completion of 'a'.

10. `path 1:([a], [b]) end ;`

Routines 'a' and 'b' operate in mutual exclusion. Either is authorized to proceed as long as requests for its execution exist. When the executing routine's request list is exhausted, either routine may start again.

11. Due to the production rules for the derestriction operation, none of the following paths are equivalent to another:

```
path 1:([a],[b]) end;
        or
path 1:(a,b) end;
path [a], [b] end;
        or
path [a], [b] end;
path 1:(a,b) end;
```

The derestriction operation should only be specified in the same path as the restriction which it is derestricting.

3.3. Syntax

The BNF syntax for Open Paths is shown below:

```
path_decl ::=
        path_expr |
        path_decl path_expr

path_list ::=
        path <list> end
```

```
list ::=
        <sequence> { , <sequence> }

sequence ::=
        <item> { ; <item> }

item ::=
        <bound> : ( <list> ) |
        [ <list> ] |
        ( <list> ) |
        <ident>

bound  ::=
        <unsgnd_int> |
        <const>
```

4. Processes

A process is a program structuring unit which has an independent execution sequence associated with it. Processes can interact and are coordinated by performing operations on synchronised shared variables. In Path Pascal, the declaration of a process is separated from its activation. A process may be declared in any block and activations of the process may be created from any body of code with scope that includes the declaration.

Processes are declared in a manner similar to standard Pascal procedures. They may possess parameters (passed by value or by reference) and may also have a size attribute. The optional size attribute is an estimate of the process's storage requirements.

4.1. Instantiation

An instance of a process is dynamically created by invoking the process name in the same manner as a procedure invocation. The creating process need not wait for the created process to terminate and continues its own execution. Each process created is allocated a run-time heap and stack from the global runtime heap. The number of bytes allocated is optionally specified by the size attribute. No mechanism is provided to abnormally terminate a process; termination occurs only when the end of a process's code body is reached.

4.2. Process Storage Considerations

Processes may themselves spawn processes. The storage from any process is acquired from the global heap of the path pascal virtual machine. It is occasionally desirable to specify a larger or smaller heap for a process than that of the default. This may need to be done due to storage used in procedures called by the process. This is done by inserting the storage requirement in words between the name of the process and the parameters (if any). An example is:

```
process bigun [50000] (arg: integer);
```

A process's storage is automatically released when a process terminates.

4.3. Process Lifetimes

The lifetime of a processes depends on both static and dynamic properties. A process remains in an block as long as any other process is referencing information in that block. Currently, only static references are detected.

If a process reaches the end of a block which is being referenced by other processes, it waits until those processes complete, at which time it releases the activation record associated with the block. If no other process references the activation record, the process need not wait. For example:

```
procedure A;
    var i : integer;

    process B;
        process C;
        begin
                delay(10);
                i := wallclock;
        end;
    begin
            C;
    end;
begin
        B;
end;
```

If the process calling procedure A were called, one activation each of processes B and C would be created. The process calling procedure A would need to wait until process C ends, but process B would not need to wait for process C, and the process calling procedure A would not need to wait for process B.

The reason for this is obvious from the body of process C. Process C references a variable in the scope of process A, but it does not reference any variables in the scope of process B. It is possible to have a dynamic reference to an activation record, by the following means:

```
process A;
    var k : integer;

    process B( var j : integer );
        begin
            j := 0;
        end;

    process C;
        begin
            B(k);
        end;

    begin
        C;
    end;
```

Process C obviously makes a reference to the variable 'k' in process A, but it is not detected by static scoping. While the reference to 'k' is not currently noted, this will be done in future releases of PPC. This leads us to the (undetected & unenforced) restriction which Experimental Path Pascal demands, namely Parameter Restriction.

4.4. Parameter Restriction

The scope of an actual parameter which is passed by reference to a process must contain scope of the process's declaration (hence storage for the parameter will exist as long as the process does). It should be noted that a call to an entry operation carries an implicit reference to the object containing the entry operations. Thus, calling a process within an object could fall under 'parameter restriction'.

4.5. Simulated Time

A process can be delayed for a fixed time interval by calling the procedure 'delay'. Its integer argument specifies how long the process is to be delayed. The number of simulated time units which have elapsed since execution began can be obtained from the parameterless integer function 'wallclock'. Wallclock was originally defined in Berkley Pascal as returning the 'time of day' in seconds from some point in the early '70s. When programs are run in 'non- simulation mode', where a call to 'delay' causes an actual delay, this meaning will hold. Currently, simulation mode is the only option possible, and wallclock returns the simulated time.

4.6. Interrupt Processes

Interrupt processes were defined in Experimental Path Pascal, but have not yet been implemented in PPC. Future releases of this reference manual will have more information regarding interrupt processes and their implementation. One aspect of interrupt process which has already been implemented is absolute variable bindings.

Bindings between variable names and absolute memory locations can be assigned by an extension of the *var* mechanism. This is intended to provide access to memory-mapped I/O devices. The name of the variable to be allocated is succeeded by the location to be assigned enclosed in square brackets. The address can be any valid Berkley Pascal integer constant. Consider the following example which sets absolute long-word 144 to 0.

```
var
    poke [ 100 ] : array[0..100] of integer;
begin
    poke [44] := 0;
end
```

4.7. Process Syntax

```
procs_decl ::=
        <procs_hdg> <block>

procs_hdg ::=
        process <id> <size_part> ; |
        process <id> <size_part> ( <formal_parm_sec> { ; <formal_parm_sec> } ) ;

size_part ::=
        [ <const> ] |
        <empty>
```

Summary

PPC is an extension of Berkley Pascal with extensions for concurrent processes, data encapsulation, absolute bind of variables and Path Expressions. The PPC compiler is written in C and produces PCCIR, the Portable C Compiler Intermediate Representation. The compiler attempts to provide debugging support through standard UNIX debuggers. The language is a re-implementation of the Experimental Path Pascal compiler developed at the University of Illinois.

Implementation details can be found in the forth-comming master thesis of the third author. The system currently runs on the DEC-VAX architecture running 4.x BSD UNIX and Sun Microsystems workstations.

References

[Ammann, et al., 76] Ammann, U., K. Nori, and C. Jacobi, "The Portable Pascal Compiler," Institut Fuer Informatik, EIDG, Technische Hochschule CH-8096, Zurich, 1976.

[Andler, 79] Andler, Sten, "Predicate Path Expressions," 6th Annual ACM Symposium on Principles of Programming Languages, San Antonio, Tex., pp. 226-236, 1979.

[Campbell & Habermann, 74] Campbell, R. H., and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," *Lecture Notes in Computer Science* (Editors G. Goos and J. Hartmanis), Vol. 16, pp. 89-102, Springer-Verlag, 1974.

[Campbell, 76] Campbell, R. H., "Path Expressions: A technique for specifying process synchronization," Ph.D. Thesis, The University of Newcastle upon Tyne, August, 1976; Also, Department of Computer Science Technical Report, University of Illinois at Urbana-Champaign, UIUCDCS-R-77-863, May, 1977.

[Campbell & Kolstad, 79a] Campbell, R. H. and R. B. Kolstad, "Path Expressions in Pascal," Fourth International Conference on Software Engineering, Munich, September 17-19, 1979.

[Campbell & Kolstad, 79b] Campbell, R. H. and R. B. Kolstad, "Practical Applications of Path Expressions to Systems Programming," ACM79, Detroit, 1979.

[Campbell & Kolstad, 80] Campbell, R. H. and R. B. Kolstad, "A Practical Implementation of Path Pascal," Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, UIUCDCS-R-80-1008, 1980.

[Dahl, et al., 68] Dahl, O. J., B. Myhrhaug, and K. Nygaard, "The Simula 67 Common Base Language," Norwegian Computer Center, Oslo, 1968.

[Flon & Habermann, 76] Flon, L. and A. N. Habermann, "Towards the Construction of Verifiable Software Systems," SIGPLAN Notices Vol. 8, No. 2, March, 1976.

[Habermann, 75] Habermann, A. N., "Path Expressions," Department of Computer Science Technical Report, Carnegie-Mellon University, June, 1975.

[Habermann, 76] Habermann, A. N., *Introduction to Operating System Design,* Science Research Associates, p. 89, 1976.

[Horton & Campbell, 80] Horton, Kurt H. and Roy H. Campbell, "PDP-11 Path Pascal Implementation Manual," Technical Report, University of Illinois at Urbana-Champaign, to be published, 1980.

[Jensen & Wirth, 75] Jensen, K. and N. Wirth, *Pascal User Manual and Report,* Springer-Verlag, New York, 1975.

[Joy, Graham & Haley, 80] William Joy, Susan Graham and Charles Haley, *Berkley Pascal User's Manual, Version 2.0* Technical Report, Dept. of Electrical Engineering and Computer Science, University of California, Berkley

[Lauer & Campbell, 75] Lauer, P. E. and R. H. Campbell, "Formal Semantics of a Class of High Level Primitives for Co–ordinating Concurrent Processes," Acta Informatica, No. 5, pp. 297–332, 1975.

[Lauer & Shields, 78] Lauer, P. E. and M. W. Shields, "Abstract Specification of Resource Accessing Disciplines: Adequacy, Starvation, Priority and Interrupts," SIGPLAN Notices, Vol. 13, Number 12, pp. 41–59, 1978.

[Miller, 78] Miller, T. J., "An Implementation of Path Expressions in Pascal," M. S. Thesis, University of Illinois, Urbana, May, 1978.

[ONERA CERT, 78] "Parallelism, Control and Synchronization Expression in a Single Assignment Language," Sigplan Notices Vol. 13, No. 1, January, 1978.

[Riddle, 76] Riddle, W. E., "Software System Modelling and Analysis," RSSM/25, Tech. Report, Department of Computer and Communication Sciences, University of Michigan, July, 1976.

[Shaw, 77] Shaw, A. C., "Software Descriptions with Flow Expressions," IEEE TSE, Vol. 4, No. 3, p. 242–254, May, 1978.

[Wirth, 77] Wirth, N., "Modula: a Language for Modular Multiprogramming," Software–Practice and Experience, Vol. 7, pp. 3–84, 1977.

Appendix A
Path Pascal Syntax

The actual syntax for PPC will be released if and when copyright agreements can be reached with UCBerkley.

Appendix B
Implementation Note

Differences between PPC and Berkley Pascal

The main differences between PPC and Berkley Pascal are as follows:

- A new type constructor 'object' is provided with synchronization and data encapsulation (see section 2).
- The 'case' construct can specify an 'otherwise' clause. The format is:

```
case_expr ::=    case ( <expr> ) of
                 case_list ;
                 otherwise : <stmt>
                 end
```

- A 'process' construct is included, allowing concurrent execution.
- Variables can be bound to hardware addresses (see section 4.6).
- A standard procedure 'delay' (see section 4.5) is introduced to delay a process for a simulated time amount of time.
- The semantics of function 'wallclock' are different in that they refer to simulated time, not real time.
- The following are new reserved words: otherwise, initially, finally, process, object, entry, path.
- The following are new system defined procedures: delay, await.
- The -C compiler flag causes a check in each procedure to ensure that the stack for the current process has not overflowed its bounds.

The differences between Experimental Path Pascal and PPC are as follows:

- The base version of Pascal is different. Since Experimental Path Pascal was based on the P4 compiler, any P4-specific functions will be lacking. Consult the Berkley Pascal Users Manual for more details concerning Berkley Pascal. Some key points are identified below.
- Name-compatible type checking is used. As an example, this may causes problems if you define instances of pointers to records as ' ^recnam '. You should declare a type 'recnampt' of type ' ^recnam ' and use that instead.

- The names for many standard functions is different. Most importantly, the old function 'time' is now called 'wallclock', in keeping with the Berkley Pascal namings. In Berkley Pascal, 'time' is a procedure used to return the alphanumeric representation of the current time. A 'pseudo-compatibility' mode can be invoked by including the provided compatibility file, which defines function 'time' in terms of 'wallclock'.

- Storage requirements are vastly different. If you have specified storage requirements for your processes, these will need to be changed to reflect the storage costs in the Berkley Pascal compiler. Typically, multiplying them by four and adding in a couple of hundred suffices. A simple procedure call takes 60 bytes. The default process size is 2044 bytes.

- Storage size for integers depends on the number of bits to represent the integer. For example, t = 0..127 would require 1 byte. Sets require as many bits (modulo word size) as the set requires. Sets can have up to 64K elements.

- The 'init' block is now a full parameterless procedures, and is called 'initially'.

- There is a 'finally' procedure to match 'initially' (see section 2.7)

- Path expressions are somewhat different in that you may specify multiple path expressions each containing at most one reference to an operation. These which are enforced in top-to-bottom in the order given (see section 3).

  typically, If you have a path of the form:

  path A1, A2, ..., An **end**

  where entry x is referenced once each in any pair Ai and Aj, i < j, you can restate this as:

  path A1, A2, ..., Ai, ..., Aj-1, Aj+1, ..., An **end**
  path Aj **end**

  Thus, the standard path for a bounded buffer, multiple-producers/consumers using shared 'input' and 'output' pointers :

  path buffers:(produce;consume), 1:(produce, consume) **end**

  becomes

  path buffers:(produce;consume) **end**
  path 1:(produce, consume) **end**

- Debugging your programs becomes a more exciting and irritating proposition. References to bad addresses and whatnot will deliver the standard UNIX "bus error(core dumped)" message. Debugging can best be accomplished using the ADB debugger at this time. Before assuming that the compiler is not working, try running your program using the -C attribute, which causes run-time checks to be turned on. Often, you will find that your process sizes are too small. For more information, see appendix E.

- The standard subroutine "malloc" is used to create processes and objects. This subroutines works most efficiently if the allocation size is 2^K - 4 bytes.

Appendix C
OPEN PATH ALGORITHMS

Open Path semantics are described below in terms of P and V operations on counting semaphores in the prologues and epilogues of the procedures, functions, and processes. The following recursive algorithm [Campbell, 77] will translate Open Paths into this P and V implementation. In general, the Path Expression to be translated will be surrounded by two strings of generated synchronisation operations which are on its left and right (L and R respectively.) Each translation rule operates on a string of the form "L M R" which represents the left, middle, and right parts of the string. The translation rule chosen to operate on M corresponds to the production rule which recognizes M (see Appendix A). The translated string consists of one or two strings of similar form and initialization code denoted by "[sx := y]."

The algorithm is initialized when Path <list> End is transformed into R=null, M=<list>, L=null. The left column of the table below shows M (assumed to be surrounded by L and R); the right columns show the new translated L, M, and R.

Transformation Table

M	new L	new M	new R	new L	new M	new R	init
<seq>,<list>	L	<seq>	R	L	<list>	R	
<item>;<seq>	L	<item>	V(s1)	P(s1)	<seq>	R	[s1:=0]
n:(<list>)	P(s2)L	<list>	R V(s2)				[s2:=n]
[<list>]	PP(c,s,L)	<list>	VV(c,s,R)				[s:=1,c:=0]
(<list>)	L	<list>	R				
<op_id>	begin L;operation;R end						

where PP(c,s,L) and VV(c, s, R) are defined as:

PP(c,s,L)	VV(c,s,R)
P(s); c := c + 1; IF c=1 THEN L; V(s)	P(s); c := c - 1; IF c=0 THEN R; V(s)

Appendix D
PROGRAMMING EXAMPLES

NETWORK

A small network simulation program patterned after [Brinch Hansen, 78] is presented below. The network is ring oriented and request-driven. Requests are sent from a processor through the network to a (probably foreign) processor, where a complementary process transmits a reply. This reply is then forwarded to the original processor. Each processor contains a single input link and a single output link. A request/response message pair circumnavigates the ring once in a normal request/respond cycle or twice if the processor attempts communication with itself. This program is presented only to compare and contrast different methods of synchronization specification, not as a solution to data transfer problems.

As presented, the program contains not only a network system, but also a simulation of the machines and physical lines. The program is somewhat shorter than Brinch Hansen's, and refers to synchronization only in the Path Expressions of the objects: semaphores (or conditions), monitors and queues are not required. The programmer can therefore simply invoke routines, knowledgeable of the fact that they are already synchronized correctly.

The program source is shown here:

```
program network(output);

const
   nmax   = 3;                    (* three nodes *)
   cmax   = 6;                    (* six channels *)
   bmax   = 3;                    (* three buffers *)

(*
 * The constants above define the network configuration
 *)

type
   node    = 1..nmax;
   channel = 1..cmax;
   channelset = set of channel;
   item    = array[1..10] of char;

   message = record
      kind: (a_request, a_response);
      link: channel;
      contents: item
   end;

(*
 * 'item' is the logical atomic data packet sent
 * between nodes. A 'message' contains routing information and the 'item'.
 *)
```

```
(*
 * The 'line' simulates the physical line between machines. Each machine
 * references two different 'line's:  one for input, one for output.
 *)

  line   = object                    (* physical line *)

     path 1:(output_to_buslink; input_from_buslink) end;
(*
 * Input must wait for output from elsewhere,
 * only a single output can occur before an input
 *)

     var mesgbuffer: message;

     entry procedure output_to_buslink(m:message);
        begin
           delay(5);
           mesgbuffer := m
        end;

     entry procedure input_from_buslink(var m:message);
        begin
           m := mesgbuffer
        end;
  end;                               (* line *)

(*
 * The 'machine' object contains all the attributes of a simulated machine.
 * these include:  'buffer' operations for the physical line; 'inputs',
 * which waits for data to be returned after a request has been sent;
 * 'outputs', which sends the data after requested;
 * 'reader', monitors traffic on line, routing messages forward or through
 * request/response mechanism;
 * 'writer', which copies messages from the output buffer to the physical line;
 * 'startmachine', forks the processes 'reader'/'writer' as initialization;
 * and finally 'receive' and 'send':  the user accessible routines to use
 * the network
 *)
```

```
machine = object

    path startmachine end;
(*
* no synchronization necessary for this initialization
*)

    type
      buffer  = object              (* handles simple queue *)

        path  bmax:( 1:(bufenter); 1:(bufretrieve)) end;
(*
* bmax outstanding requests (namely 'bufenter's) may exist,
* 'bufenter's must precede 'bufretrieve's.
*)

        var iobuffer: array[1..bmax] of message;
          inpp, outp: 1..bmax;

        entry procedure bufenter(m:message);
          begin
            iobuffer[inpp] := m;
            inpp := (inpp mod bmax) + 1
          end;

        entry procedure bufretrieve(var m:message);
          begin
            m := iobuffer[outp];
            outp := (outp mod bmax) + 1
          end;

        initially;
            begin
          inpp := 1;
          outp := 1
        end;
      end;                          (* buffer *)
(*
* Only the Path Expression synchronizes
* the buffer code.
*)
```

```
        inputs = object              (* handle inputting messages *)

           path response_received; response_wait end;
(*
 * 'response_wait' will not continue until 'response_received' is finished.
 * it then merely copies the message from the line monitor.
 *)

           var mesgcontents: item;

           entry procedure response_received(cont: item);
              begin
                 mesgcontents := cont
              end;

           entry procedure response_wait(var cont: item);
              begin
                 cont := mesgcontents
              end;
        end;                         (* inputs *)

        outputs = object             (* handles message outputs *)

           path request_received; build_mesg end;
(*
 * 'build_mesg' may not be executed until 'request_received' is complete
 *)

           entry procedure build_mesg(c:channel; info:item;
                    var mesg:message);
              begin
                 mesg.kind := a_response;
                 mesg.link := c;
                 mesg.contents := info;
              end;

           entry procedure request_received;
              begin  end;
(*
 * this procedure is empty as no code is required, only a 'signal' for
 * the Path expression to process.
 *)

        end;                         (* outputs *)
```

```
    var buf: buffer;
      inp: array [channel] of inputs;
      out: array [channel] of outputs;
(*
 * logical channels are used for communication. each machine has a different
 * set of input and output channels.
 *)

    process reader(inpset, outset:channelset; inline:line);
                    (* handle all messages from line *)
      var m: message;

      begin
        repeat
          inline.input_from_buslink(m); (* get message from line *)
          if (m.kind = a_response) and (m.link in inpset)
                        (* response for me? *)
            then inp[m.link].response_received(m.contents)
          else
            if (m.kind = a_request) and (m.link in outset)
                        (* request for me? *)
              then out[m.link].request_received
            else
              buf.bufenter(m) (* pass message on *)
        until false
      end;                      (* reader process *)

    process writer(outline:link);       (* put messages onto line *)
      var m: message;

      begin
        repeat
          buf.bufretrieve(m);
          outline.output_to_buslink(m)
        until false
      end;                      (* writerprocess *)

    entry procedure startmachine(who: node; inpset, outset:channelset;
          inline, outline: line);
      begin
        reader(inpset, outset, inline);
        writer(outline)
      end;
```

```
(*
 * user called procedures:
 *)

     procedure  receive(c:channel; var v:item);

       var mesg: message;

       begin
         mesg.kind := a_request;
         mesg.link := c;
         buf.bufenter(mesg);        (* request mesg *)
         inp[c].response_wait(v)    (* wait for response *)
       end;

     procedure  send(c: channel; info:item);

       var mesg: message;

       begin
         out[c].build_mesg(c, info, mesg); (* build mesg after reqst *)
         buf.bufenter(mesg)         (* send mesg along *)
       end;

(*
 * each machines's code would go here:  it would be invoked by startmachine
 *)

  end;                              (* machine *)

(*
 * finally, it is necessary to specify the physical lines between the
 * machines
 *)

var machines: array [node] of machine;
  lines: array [node] of line;

begin
  machines[1].startmachine(1, [2,3], [1,4], lines[3], lines[2]);
  machines[2].startmachine(2, [1,6], [2,5], lines[1], lines[3]);
  machines[3].startmachine(3, [4,5], [3,6], lines[2]. lines[1])
end.
```

DINING PHILOSOPHERS

The well known problem of the dining philosophers involves a set of five philosophers whose activities in life are eating and thinking. Each philosopher thinks for a while, eats, thinks, eats and so on. The philosophers share a unique dining arrangement: though two utensils are required for a philosopher to eat, the five dining places are located around a circular table with only one utensil on the right of each dining place. Therefore, the philosophers must share utensils. The problem involves the scheduling of the philosophers so that no philosopher attempts to begin eating when his utensils are not available. The Path Pascal solution to this problem is different from many in that no explicit queues are needed. Each philosopher is a process attempting to use the 'fork' objects. Paths synchronise access and prevent deadlocks from occurring. Note that only simple synchronisation statements are given (e.g., only four philosophers eating at a time, only one using each fork). The rest of the program specifies the logic of thinking and eating.

```
program phils(output);

const   nphilosophers = 5;
        maxindex = 4;          (* nphilosophers - 1 *)

type    diner = 0..maxindex;

var     i: integer;
        table: object
                path maxindex:(starteating; stopeating) end;
                var     fork: array [diner] of
                        object
                                path 1:(pickup; putdown) end;
                                entry procedure pickup;  begin end;
                                entry procedure putdown;  begin end;
                        end;

                entry procedure starteating(no: diner);
                        begin
                                fork[no].pickup;
                                fork[(no+1) mod nphilosophers].pickup
                        end;

                entry procedure stopeating(no: diner);
                        begin
                                fork[no].putdown;
                                fork[(no+1) mod nphilosophers].putdown;
                        end;
                end;            (* table *)
```

```
process philosopher(mynum: diner);
        begin
                repeat
                        delay(ran(seed));
                        table.starteating(mynum);
                        delay(ran(seed));
                        table.stopeating(mynum);
                until false;
        end;

begin
        for i:= 0 to maxindex do philosopher(i)
end.
```

BUFFER MANAGEMENT

A simple ring buffer implementation is shown below:

```
program buffering(output);

const  bufsize = 32;
       maxbuf = 31;

type   buffer = object                          (* buffers i/o *)

       path bufsize: (1: (fill); 1: (empty)) end;

       type   bufrange = 0..maxbuf;
              bufarray = array[bufrange] of char;

       var    inptr, outptr: bufrange;
              buf: bufarray;

       entry procedure fill(ch: char);
              begin
                     buf[inptr] := ch;
                     inptr := (inptr+1) mod bufsize
              end;

       entry procedure empty(var ch: char);
              begin
                     ch := buf[outptr];
                     outptr := (outptr+1) mod bufsize
              end;

       initially; begin inptr := 0; outptr := 0 end
   end;
```

Two routines are provided, 'fill' and 'empty'. Note that the routines are very terse: only information relating to the actually changing of pointers and data is presented. All synchronization and restriction information is described by the Path Expression, which assures mutual exclusion for each routine and places a maximum on the buffer size. Attempts to exceed the buffer size are not allowed to proceed until an element is removed from the buffer.

BIBLIOGRAPHY

[Brinch Hansen, 78] Brinch Hansen, P., "Network: A Multiprocessor Program," IEEE Trans. Software Eng., Vol. SE–4, No. 3, pp. 194–199, May, 1978.

Appendix E
Debugging Aids

To aid programmers in debugging, certain information regarding the structure of the Path Pascal run-time system is needed, as well as information regarding naming conventions for processes and object entry routines.

Namings

With each object type is associated several code bodies. These code bodies must be named internally to reference them. In the current Berkley scheme, nested procedures have nested names using the procedure names. That is, if you have the following:

```
Procedure A;
    Procedure B;
        Procedure C; end
    end
end
```

Then the code body for procedure 'A' will have the name '_A', the body for procedure 'B' will be named '_A_B' and that for procedure 'C' will be named '_A_B_C'. Additionally, each procedure named 'X' has a name '_X' used for procedure references passed as parameters. So, we also have symbols '__A', '__A_B' and '__A_B_C'. You can more or less ignore these.

For a procedure, there is a one-to-one mapping for names and code-bodies. For objects declared in types, this is also true. The name of the type is the name of the object. However, an object can be declared as an 'anonymous type' in a var declaration. Thus, in something like:

```
var
    a, b : object ... end
```

what should the name of the code bodies associated with the object be? It turns out that it is easiest, from the view point of the compiler, to associate an arbitrary 'object number' with the object.

Associated with each object is a procedure '$init' which allocates memory for the object, initializes the semaphores, calls '$init' on any objects nested within this object and calls the procedure '$initially' (if it exists). Similarly, there is a procedure '$fini', which calls the procedure '$finally' (if it exists), calls '$fini' on any nested objects and deallocates the memory for the object.

The '$initially' and '$finally' procedures are the names given to the initially and finally procedures within an object. They are only present if the corresponding routine was declared in the object.

All operations prepend their name with the objects name. As an example, consider.

```
procedure A
type
    B = object
        var i : integer;

        path C end;

        entry procedure C; begin end;

        initially; begin C end;
```

```
        end;

var
    D : object
        var i : integer;

        path E end;

        entry procedure E; begin end;

        end;

    begin
    end
```

In the above, we have the following names: _A, _A_B$obj_C, _A_B$obj_$initially, _A_B$obj_$init, _A_B$obj_$fini, _A__$obj00001_D, _A__$obj00001_$init _A__$obj00001_$fini.

Processes also present a little bit of a naming problem, but much less troublesome than objects. Essentially, we would like a process call to be equivilant to a procedure call. We do this by the following construct:

```
Process A;
begin
end;
```

is expanded as:

```
_A:
    set up parameters for run-time-system;
    using _a as the process start label.
    call 'process-create' routine
    return
_a:
    actual process code
    call 'kill process' routine
```

The 'actual process code' is preceded by a name which is the same as the process name, but reversed in case (that is A -> a, b -> B, etc).

Object Structures

Calls to entry procedures causes an implicit reference to the context for the given object. Berkley Pascal uses a 'display list' stored in _disply to handle static nestings. Object references use this same display list to handle references to variables within objects. The display list is a per-process data structure, but the context switching routines correctly saves it.

On every call to an object entry, the base address for the object must be passed to the entry procedure so that it may fix up the display list. This is pushed as an invisible first parameter on any entry call. Thus, a call such as:

```
A.entryname(1,2,3)
```

is really processed as:

```
entryname(addrof(A), 1, 2, 3)
```

You need to be aware of this when using ADB to debug your object entries.

Run Time System

The run time system is written in C. The interface between the run time system and the Path Pascal program is managed by two procedure: _rts_call and _rts_exec. When a process wishes to request a service from the run time system, it pushes the parameters and the service number and then calls p. ocedure _rts_call. This saves the context for this process and starts up the context for the run–time–system.

When the run–time–system wishes to restart a Path Pascal process from where it was last suspended, it calls _rts_exec with the pointer to the Process Control Block for that process.

Debugging information can be turned on by calling the routine "rtsdebugging" with a non–zero integer parameter. The higher the number, the more debugging information. This can be called from Path Pascal if you define:

```
procedure rtsdebugging ( i : integer ); external;
```

This information is not exactly the most clear or informative information possible, but it is a good way to untangle deadlocks and what not.

Future debugging aids

In the future, we plan to provide a post–mortem dump analys s routine as well as extensions to allow DBX to work with Path Pascal.