Palo Alto Research Center

# Computer Science Laboratory Technical Report Digest: 1973-1990

Compiled and edited by: Nancy Freige

XEROX

# Computer Science Laboratory Technical Report Digest: 1973-1990

Compiled and edited by: Nancy Freige

CSL-91-12 November 1991 [P91-00141]



**Abstract:** This digest contains a summary of technical reports published by the Computer Science Laboratory of the Xerox Palo Alto Research Center, for the period 1973 through 1990. Entries are sorted by publication date, and contain the report title, publication number, author(s) and abstract.

Reports preceded by an asterisk (*) in the table of contents are out of print and no longer available. All other reports can be obtained by writing to the CSL Receptionist at the address below.

**CR Categories and Subject Descriptors:** A.2 [Reference]

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

# PREFACE

The Computer Science Laboratory has a long history of research focused on fundamental issues in the design, development, and use of distributed computer systems. As chronicled in this digest of abstracts, research in the Laboratory has spanned a broad range of topics and led to fundamental contributions in many areas including programming languages and environments, computer graphics and imaging, personal computers, local area networks, computer communication protocols, programming language theory, computational geometry, and distributed multi-media computing.

Today, the mission of the Computer Science Laboratory's research is to create a conceptual and empirical foundation for advanced computing systems. The vision at the heart of this mission is one in which integrated, interoperable, readily available, powerful computational devices invisibly enhance human activities. Referred to as *ubiquitous computing*, this vision is intended to define a new computing paradigm as well as provide a foundation for Xerox initiatives in the next generation of document processing systems and services. Within this context the program focuses on core computer science issues in the development of portable and interoperable systems, distributed systems and collaborative technologies, information storage and retrieval, computation methods, and computational hardware and architecture. The objective throughout is to understand the fundamental principles and underpinnings of ubiquitous computing.

Mark Weiser, Manager
John R. White, Associate Manager
Computer Science Laboratory


MURPHY 89

# TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

**1985**



**1986**

**CSL-73-1 May 1973**

## AN INTERACTIVE PROGRAM VERIFIER
**by L. Peter Deutsch**

Program verification refers to the idea that the intent or effect of a program can be stated in a precise way that is not a simple "rewording" of the program itself, and that one can prove (in the mathematical sense) that a program actually conforms to a given statement of intent. This thesis describes a software system which can verify (prove) some non-trivial programs automatically.

The system described here is organized in a novel manner compared to most other theorem-proving systems. It has a great deal of specific knowledge about integers and arrays of integers, yet it is not "special-purpose", since this knowledge is represented in procedures which are separate from the underlying structure of the system. It also incorporates some knowledge, gained by the author from both experiment and introspection, about how programs are often constructed, and uses this knowledge to guide the proof process. It uses its knowledge, plus contextual information from the program being verified, to simplify the theorems dramatically as they are being constructed, rather than relying on a super-powerful proof procedure. The system also provides for interactive editing of programs and assertions, and for detailed human control of the proof process when the system cannot produce a proof (or counter-example) on its own.

**CSL-73-2 August 1973**

## NEW PROGRAMMING LANGUAGES FOR AI RESEARCH
**by Daniel G. Bobrow and Bertram Raphael***

New directions in Artificial Intelligence research have led to the need for certain novel features to be embedded in programming languages. This paper gives an overview of the nature of these features, and their implementation in four principal families of AI languages: SAIL; PLANNER/CONNIVER; QLISP/INTERLISP; and POPLER/POP-1. The programming features described include: new data types and accessing

mechanisms for stored expressions; more flexible control structures, including multiple processes and backtracking; pattern matching to allow comparison of data item with a template, and extraction of labeled subexpressions; and deductive mechanisms which allow the programming system to carry out certain activities including modifying the data base and deciding which subroutines to run next using only constraints and guidelines set up by the programmer.

*Stanford Research Institute

**CSL-73-3 August 1973**

**THE IMPLEMENTATION OF NLS ON A MINICOMPUTER**
**by James G. Mitchell**

This technical report covers the research performed at Xerox Palo Alto Research Center (PARC) for the period June 30, 1972 to July 1, 1973 under Contract Number DAHCI5 72 C 0223 with the Advanced Research Projects Agency, Information Processing Techniques Office. The research covers initial studies and evaluation of transferring a large, display-oriented documentation system (the NLS system developed at Stanford Research Institute) to a minicomputer system and a protocol for accessing NLS over the ARPANET.

**CSL-73-4 December 1973**

**OMNIGRAPH: SIMPLE TERMINAL-INDEPENDENT GRAPHICS SOFTWARE**
**by Robert F. Sproull**

This paper describes a graphics subroutine package for driving a number of different display devices with any of three different programming languages. The Omnigraph system is designed for routine graphics applications, not for high-performance terminals. The success of the design is largely due to the modest aims of the routines and to the particularly simple framework chosen for the graphics facilities.

This paper cites a number of design errors in the initial Omnigraph routines, and suggests improvements. The Omnigraph Reference Manual is reprinted as an appendix.

**CSL-74-1 January 1974**

**A POSTMORTEM FOR A TIME SHARING SYSTEM**
**by Howard Ewing Sturgis**

This thesis describes a time sharing system constructed by a project at the University of California, Berkeley Campus, Computer Center. The project was of modest size, consuming about 30 man years. The resulting system was used by a number of programmers. The system was designed for a commercially available computer, the Control Data 6400 with extended core store. The system design was based on several fundamental ideas, including:

- specification of the entire system as an abstract machine,
- a capability based protection system,
- mapped address space,
- and layered implementation.

The abstract machine defined by the first implementation layer provided 8 types of abstractly defined objects and about 100 actions to manipulate them. Subsequent layers provided a few very complicated additional types. Many of the fundamental ideas served us well, particularly the concept that the system defines an abstract machine, and capability based protection. However, the attempt to provide a mapped address space using unsuitable hardware was a disaster. This thesis includes software and hardware proposals to increase the efficiency of representing an abstract machine and providing capability based protection. Also included is a description of a crash recovery consistency problem for files which reside in several levels of storage, together with a solution that we used.

**CSL-74-2 May 1974**

**ON DATA-LIMITED AND RESOURCE LIMITED PROCESSES**
**by Donald A. Norman and Daniel G. Bobrow**

This paper analyzes the effect on performance when several active processes compete for limited processing resources. The principles discussed in this paper show that conclusions about the interactions among psychological processes must be made with caution, and some existing assumptions may be unwarranted. When two (or more) processes use the same resources at the same time, they may both interfere with one another, neither may interfere with the other, or one may interfere with a second without any interference from the second process to the first. The important principles are that a process can be limited in its performance either by limits in the amount of available processing resources (such as memory or processing effort) or by limits in the quality of the data available to it. Competition among processes can affect a resource-limited process, but not a data-limited one. If a process continually makes preliminary results available even before it has completed all its operations, then it is possible to compute performance-resource operating characteristics that show how processes interact. A number of experiments from the psychological literature are examined according to these processing principles, resulting in some new interpretations of interactions among competing psychological processes.

**CSL-74-3 December 1974; Revised March 1975**

**INTRODUCING ITERATION INTO THE PURE LISP THEOREM PROVER**
**by J. Strother Moore**

It is shown how the LISP iterative primitives PROG, SIETQ, GO, and RETURN may be introduced into the Boyer-Moore method for automatically verifying Pure LISP programs. This is done by extending some of the previously described heuristics for dealing with recursive functions. The resulting verification procedure uses structural induction to handle both recursion and iteration. The procedure does not actually distinguish between the two and they may be mixed arbitrarily. For

example, since properties are stated in terms of user-defined functions, the theorem prover will prove recursively specified properties of iterative functions. Like its predecessor, the procedure does not require user-supplied inductive assertions for the iterative programs.

**CSL-75-1 January 1975**

**ON THE PROBLEM OF UNIFORM REFERENCES TO DATA STRUCTURES**
**by Charles M. Geschke and James G. Mitchell**

The cost of a change to a large software system is often primarily a function of the size of the system rather than the complexity of the change. One reason for this is that programs which access some given data structure must operate on it using notations which are determined by its exact representation. Thus, changing how it is implemented may necessitate changes to the programs which access it. This paper develops a programming language notation and semantic interpretations which allow a program to operate on a data object in a manner which is dependent only on its logical or abstract properties and independent of its underlying concrete representation.

**CSL-75-2 April 1975**

**COMPUTATIONAL LOGIC: STRUCTURE SHARING AND PROOF OF PROGRAM PROPERTIES, PART II**
**by J. Strother Moore**

This paper describes a program which automatically proves a wide variety of theorems about functions written in a subset of pure LISP. Features of this program include: The program is fully automatic, requiring no information from the user except the LISP definitions of the functions involved and the statement of the theorem to be proved. No inductive assertions are required from the user. The program uses structural induction when required, automatically generating its own induction formulas. All relationships in the theorem are expressed in terms of user defined LISP functions, rather than a second logical language. The system employs no built-in

information about any non-primitive function. All properties required for any function involved in a proof are derived and established automatically. The program is capable of generalizing some theorems in order to prove them; in doing so, it often generates interesting lemmas. The program can write new, recursive LISP functions automatically in attempting to generalize a theorem. Finally, the program is very fast by theorem proving standards, requiring around 10 seconds per proof.

**CSL-75-3 April 1975**

**A SPACE-ECONOMICAL SUFFIX TREE CONSTRUCTION ALGORITHM**
**by Edward M. McCreight**

The first section presents a new algorithm for constructing auxiliary digital search trees to aid in exact-match substring searching. This algorithm has the same asymptotic running-time bound as previously published algorithms, but is more economical in space. The second section discusses some implementation considerations. The third section presents new work dealing with how to modify these search trees in response to incremental changes in the strings they index (the update problem).

**CSL-75-4 July 1975**

**SOME PRINCIPLES OF MEMORY SCHEMATA**
**by Daniel G. Bobrow and Donald A. Norman***

This paper deals with two related issues about memory: access and processing. Consideration of the properties of human memory lead us to suggest that memory is organized into structural units: *schemata*. We suggest that memory schemata refer to one another by means of *context dependent descriptions* that specify the referent unambiguously only with respect to a particular context. We argue that this method of memory reference has a number of desirable features for any intelligent memory system. For one, it leads automatically to metaphorical and analogical match of

memory structures. For another, it produces systems that are robust and relatively insensitive to errors.

Consideration of systems which have limits on processing resources leads to some basic principles of processing that apply to memory structures. The quality of output of some processes is limited by the quality of data available to them (these are *data-limited processes*). The quality of the output of other processes is limited by the amount of processing resources available to them (these are *resource-limited processes*). All processes are either data-limited or resource-limited. We suggest that the overall system is driven from two levels--by the data, and by concepts or hypotheses of what is expected. These considerations of processing principles provide some useful interpretations of psychological phenomena, and suggest possible useful computational models for artificial systems.

*Department of Psychology, University of California at San Diego

**CSL-75-5 July 1975**

**DIMENSIONS OF REPRESENTATION**
**by Daniel G. Bobrow**

A set of questions is presented concerning representations of knowledge. The questions are organized in terms of a framework in which knowledge of a world-state is derived by mapping the world to a knowledge base. The dimensions of representation are defined in terms of design issues which must be faced or finessed in any representation. Issues considered include correspondence between operations on the world and on the knowledge base, organization of the mapping process, inference which would make explicit knowledge which would otherwise be implicit, philosophy and mechanisms of access to elements of the knowledge base, pattern matching in knowledge processing, types of self-awareness an understander system might have, and use of multiple representations. These dimensions are illustrated with examples from the literature. The differences between

analogical and propositional representations are illuminated by consideration along the multiple dimensions of representation.

**CSL-75-6 July 1975**

**SUBGOAL INDUCTION**
**by James H. Morris, Jr., and Ben Wegbreit**

A new proof method, subgoal induction, is presented as an alternative or supplement to the commonly used inductive assertion method. Its major virtue is that it can often be used to prove a loop's correctness directly from its input-output specification without the use of an invariant. The relation between subgoal induction and other commonly used induction rules is explored and, in particular, it is shown that subgoal induction can be viewed as a specialized form of computation induction. Finally, a set of sufficient conditions are presented which guarantee that an input-output specification is strong enough for the induction step of a proof by subgoal induction to be valid.

**CSL-75-7 November 1975**

**ETHERNET: DISTRIBUTED PACKET SWITCHING FOR LOCAL COMPUTER NETWORKS**
**by Robert M. Metcalfe and David R. Boggs**

Ethernet is a branching broadcast communication system for carrying digital data packets among locally distributed computing stations. The packet transport mechanism provided by Ethernet has been used to build systems which can be viewed as either local computer networks or loosely coupled multiprocessors.

An Ethernet's shared communication facility, its Ether, is a passive broadcast medium with no central control. Coordination of access to the Ether for packet broadcasts is distributed among the contending transmitting stations using controlled statistical arbitration. Switching of packets to their destinations on the Ether is distributed among the receiving stations using packet address recognition.

Design principles and implementation are described based on experience with an operating Ethernet of 100 nodes along a kilometer of coaxial cable. A model for estimating performance under heavy loads and a packet protocol for error-controlled communication are included for completeness.

**CSL-75-8 September 1975**

**GOAL-DIRECTED PROGRAM TRANSFORMATION**
**by Ben Wegbreit**

Program development often proceeds by transforming simple, clear programs into complex, involuted, but more efficient ones. This paper examines ways this process can be rendered more systematic. We show how analysis of program performance, partial evaluation of functions, and abstraction of recursive function definitions from recurring subgoals can be combined to yield many global transformations in a methodical fashion. Examples are drawn from compiler optimization, list processing, very high level languages, and APL execution.

**CSL-76-1 July 1976**

**A FAST STRING SEARCHING ALGORITHM**
**by Robert S. Boyer* and J. Strother Moore***

An algorithm is presented that searches for the location, **i**, of the first occurrence of a character string, **pat**, in another string, **string**. During the search operation, the characters of **pat** are matched starting with the last character of **pat**. The information gained by starting the match at the end of the pattern often allows the algorithm to proceed in large jumps through the text being searched. Thus, the algorithm has the unusual property that, in most cases, not all of the first **i** characters of **string** are inspected. The number of characters actually inspected (on the average) decreases as a function of the length of **pat**. For a random English pattern of length 5, the algorithm will typically inspect **i/4** characters of **string** before finding a match at **i**. Furthermore, the algorithm has been implemented so that (on the average) fewer than **i + patlen** machine instructions are executed. These

conclusions are supported with empirical evidence and a theoretical analysis of the average behavior of the algorithm. The worst case behavior of the algorithm is linear in **l + patlen**, assuming the availability of array space for tables linear in **patlen** plus the size of the alphabet.

*Stanford Research Institute, Computer Science Group

**CSL-76-2 July 1976**

**CONSTRUCTIVE METHODS IN PROGRAM VERIFICATION**
**by Ben Wegbreit**

Most current approaches to mechanical program verification transform a program, and its specifications into first order formulas and try to prove these formulas valid. Since the first order predicate calculus is not decidable, such approaches are inherently limited. This paper proposes an alternative approach to program verification: Correctness proofs are constructively established by *proof justifications* written in an algorithmic notation. These *proof justifications* are written as part of the program, along with the executable instructions and correctness specifications. A notation is presented in which instructions, specifications, and justifications are neatly interwoven. The justifications establish the collection between the instructions and specifications; they document the reasoning on which the correctness is based. Programs so written may be verified by proving the truth of quantifier-free logical formulas.

**CSL-76-3 July 1976**

**THE ANALYSIS OF HASHING ALGORITHMS**
**by Leonidas J. Guibas**

In this thesis we relate the performance of hashing algorithms to the notion of *clustering*, that is the pile-up phenomenon that occurs because many keys may probe the table locations in the same sequence. We will say that a hashing technique exhibits *k-ary clustering* if the search for a key begins with k independent random probes and the subsequent sequence of probes is completely determined by the location of the k

initial probes. Such techniques may be very bad; for instance, the average number of probes necessary for insertion may grow linearly with the table size. However, on the average (that is if the permutations describing the method are randomly chosen), k-ary clustering techniques for $k > 1$ are very good. In fact the average performance is asymptotically equivalent to the performance of *uniform probing*, a method that exhibits no clustering and is known to be optimal in a certain sense.

Perhaps the most famous among tertiary clustering techniques is *double hashing*, the method in which we probe the hash table along arithmetic progressions where the initial element and the increment of the progression are chosen randomly and independently depending only on the key K of the search. We prove that double hashing is also asymptotically equivalent to uniform probing for load factors $\alpha$ not exceeding a certain constant $\alpha_0 = .31...$ . Our proof method has a different flavor from those previously used in algorithmic analysis. We begin by showing that the tail of the hypergeometric distribution a fixed percent away from the mean is exponentially small. We use this result to prove that random subsets of the finite ring of integers modulo m of cardinality $\alpha m$ have always nearly the expected number of arithmetic progressions of length k, except with exponentially small probability. We then use this theorem to start up a process (called the *extension process*) of looking at snapshots of the table as it fills up with double hashing. Between steps of the extension process we can show that the effect of clustering is negligible, and that we therefore never depart too far from the truly random situation.

**CSL-76-4 July 1976**

**AN OVERVIEW OF KRL, A KNOWLEDGE REPRESENTATION LANGUAGE**
**by Daniel G. Bobrow and Terry Winograd***

This paper describes KRL, a Knowledge Representation Language designed for use in understander systems. It outlines both the general concepts which underlie our research and the details of KRL-0, an experimental implementation of some of these concepts. KRL is an attempt

to integrate procedural knowledge with a broad base of declarative forms. These forms provide a variety of ways to express the logical structure of the knowledge, in order to give flexibility in associating procedures (for memory and reasoning) with specific pieces of knowledge, and to control the relative accessibility of different facts and descriptions. The formalism for declarative knowledge is based on structured conceptual objects with associated descriptions. These objects form a network of memory units with several different sorts of linkages, each having well-specified implications for the retrieval process. Procedures can be associated directly with the internal structure of a conceptual object. This procedural attachment allows the steps for a particular operation to be determined by characteristics of the specific entities involved.

The control structure of KRL is based on the belief that the next generation of intelligent programs will integrate data-directed and goal-directed processing by using multi-processing. It provides for a priority-ordered multi-process agenda with explicit (user-provided) strategies for scheduling and resource allocation. It provides procedure directories which operate along with process frame works to allow procedural parametrization of the fundamental system processes for building, comparing, and retrieving memory structures. Future development of KRL will include integrating procedure definition with the descriptive formalism.

*Stanford University AI Laboratory

**CSL-76-5 September 1976**

**THE INTERLISP VIRTUAL MACHINE SPECIFICATION**
**by J. Strother Moore**

The INTERLISP Virtual Machine is the environment in which the INTERLISP System is implemented. It includes such abstract objects as "Literal Atoms," "List Cells," "Integers," etc., the basic LISP functions for manipulating them, the underlying program control and variable binding mechanisms, the input/output facilities, and interrupt processing facilities. In order to implement the INTERLISP System (as described in *The INTERLISP Reference Manual* by W. Teitelman, et al.) on some physical machine, it is only necessary to implement the

INTERLISP Virtual Machine, since Virtual Machine compatible source code for the rest of the INTERLISP System can be obtained from publicly available files. This document specifies the behavior of the INTERLISP Virtual Machine from the implementor's point of view. That is, it is an attempt to make explicit those things which must be implemented to allow the INTERLISP System to run on some machine.

**CSL-76-6 October 1976**

**EARLY EXPERIENCES WITH MESA**
**by Charles Geschke, James H. Morris and Ed Satterthwaite**

The experiences of Mesa's first users -- primarily its implementors -- are discussed, and some implications for Mesa and similar programming languages are suggested. The specific topics addressed are:

- module structure and its use in defining abstractions,
- data-structuring facilities in Mesa,
- equivalence algorithm for types and type coercions,
- benefits of the type system and why it is breached occasionally,
- difficulty of making the treatment of variant records safe.

**CSL-76-7 December 1976**

**META-PROGRAMMING: A SOFTWARE PRODUCTION METHOD**
**by Charles Simonyi**

This thesis describes an organizational schema, designed to yield very high programming productivity in a simplified task environment which excludes scheduling, system design, documentation, and other engineering activities. The leverage provided by high productivity can, in turn, be used to simplify the engineering tasks. Difficulty of communications within a production team, caused by the inherently rapid creation of problem specific *local language*, is posited as the major obstacle to the improvement of productivity. The thesis proposes a combination of ideas for simplifying

communications between programmers. Meta-programs are informal, written communications, from the meta-programmer, who creates the local language, to technicians who learn it and actually write the programs.

The abstract notion of local language is resolved into the questions: What are the objects that should be named, and what should their names be? The answers involve the concept of painted types (related to types in programming languages), and naming conventions based on the idea of identifying objects by their types. A method of state vector syntax checking for debugging the programs produced in the high productivity environment is described.

Descriptions of the relationships or contrasts between the meta-programming organization and the relevant software engineering concepts of high level languages, egoless programming, structured programming, Chief Programmer Teams, and automatic program verification are also given.

To verify the predictions of the meta-programming theory, a series of experiments were performed. In one of the projects, three programs were produced from the same specifications, by three different groups in a controlled experiment. During the longest experiment 14,000 lines of code were written, at an average rate of 6.12 lines/man-hour. The controlled experiments showed that comparable results can be obtained by different persons acting as meta-programmers. The difficult experimental comparisons of the meta-programming and conventional organizations, however, yielded interesting, but inconclusive, results.

**CSL-77-1**

**SCHEMES: A HIGH LEVEL DATA STRUCTURING CONCEPT**
**by James G. Mitchell and Ben Wegbreit**

In recent years, programming languages have provided better constructs for data type definitions and have placed increasing reliance on type machinery for protection, modularization, and abstraction. This paper introduces several new constructs which further these ends. Types may be defined as similar to

existing types, *extended* by additional properties. *Schemes* are type-parameterized definitions. For example, symbol tables and symbol table operations can be defined as a scheme with the key and value types as parameters; an instantiation of the scheme implements a specific type of symbol table. Because new types are typically defined along with other related types, an instantiated scheme may *export* a set of new types. A set of schemes with a common name and common external behavior can be viewed as alternative implementations of an *abstraction*. Parameter specifications associated with each scheme are used to select the appropriate implementation for each use.

**CSL-77-2 July 1977**

**STRATEGY CONSTRUCTION USING A SYNTHESIS OF HEURISTIC AND DECISION-THEORETIC METHODS**
**by Robert F. Sproull**

This report describes a framework for constructing plans, or *strategies*, in which aspects of mathematical decision theory are incorporated into symbolic problem-solving techniques currently dominant in artificial intelligence. The utility function of decision theory is used to reveal tradeoffs among competing strategies for achieving various goals, taking into account reliability, the complexity of steps in the strategy, the value of the goal, and so forth. The utility function aids searching for good strategies, acquiring a world model, allocating planning effort, and organizing a hierarchical problem-solving system.

A problem-solving system that prepares travel itineraries is presented as a case study in integrating the techniques of decision theory and artificial intelligence. The system uses a model of the traveler's utility to organize a search for good solutions. The hierarchical structure of the search narrows the search by finding crude plans and then further refining them.

A central observation of this work is that locating an *optimal strategy* is not the proper procedure when the costs of the planning itself are taken into account. Instead, we desire to engage in *optimal planning*, in which the total expenditure of

effort to find and execute the solution is in some sense optimal.

**CSL-77-3 March 1977**

**A DISPLAY ORIENTED PROGRAMMER'S ASSISTANT**
**by Warren Teitelman**

This paper continues and extends previous work by the author in developing systems which provide the user with various forms of explicit and implicit assistance, and in general cooperate with the user in the development of his programs. The system described in this paper makes extensive use of a bit map display and pointing device (a mouse) to significantly enrich the user's interactions with the system, and to provide capabilities not possible with terminals that essentially emulate hard copy devices. For example, any text that is displayed on the screen can be pointed at and treated as input, exactly as though it were typed, i.e., the user can say use *this* expression or *that* value, and then simply point. The user views his programming environment through a collection of display windows, each of which corresponds to a different task or context. The user can manipulate the windows, or the contents of a particular window, by a combination of keyboard inputs or pointing operations. The technique of using different windows for different tasks makes it easy for the user to manage several simultaneous tasks and contexts, e.g., defining programs, testing programs, editing, asking the system for assistance, sending and receiving message, etc. and to switch back and forth between these tasks at his convenience.

**CSL-77-4 June 1977**

**A NECESSARY AND SUFFICIENT CONDITION FOR THE EXISTENCE OF HOARE LOGICS**
**by Richard J. Lipton**

A necessary and sufficient condition is obtained for the existence of Hoare Logics for a large class of programming languages. In addition, Cook's concept of expressiveness is shown to be a very powerful restriction on the assertion language.

**CSL-78-3 September 1978**

**EMPIRICAL ESTIMATES OF PROGRAM ENTROPY**
**by Richard E. Sweet**

We wish to investigate compact representation of object programs, therefore we wish to measure entropy, the average information content of programs. This number tells how many bits, on the average, would be needed to represent a program in the best possible encoding. A collection of 114 MESA programs, comprising approximately a million characters of source text, is analyzed. For analysis purposes, the programs are represented by trees, obtained by taking the parse trees from the compiler before the code generation pass and merging some of the symbol table information into them.

A new definition is given for a Markov source where the concept of "previous" is defined in terms of the tree structure, and this definition is used to model the MESA program source. The lowest entropy value for these Markov models is 1.7 bits per tree node, assuming dependencies of each node on its grandfather, father, and elder brother (order 3). These numbers compare with an approximate 10 bits per node required for a naive encoding, and an equivalent of 3.2 bits per node of code generated by the existing compiler. Motivated by sample set limitations for higher order models, we derive an entropy formula in which the order is non-uniform.

The non-uniform entropy formulas are particularly suited to trees, where we can now speak of conditional probabilities in terms of patterns or arbitrarily shaped contexts around a node. A method called pattern refinement is presented whereby patterns are "grown", i.e., the set of nodes matching an existing pattern is divided into those matching a larger pattern and those remaining. A proof is given that the process always leads to a lower estimate unless the old and new patterns induce exactly the same conditional probabilities. The result of applying this technique to the sample was an estimate of 1.6 bits per node. Further application would reduce this number even more.

Analytic solutions for the error bounds in approximating the entropy of a Markov source are very difficult to obtain, so an experimental approach is used to gauge a confidence figure for the estimate. These calculations suggest that a more accurate estimate would be 1.8 bits per node, with a standard deviation of 13%. This corresponds to an entropy of .54 bits per character of source program.

The methods of this thesis can be used both to define a bound for code compression and to evaluate existing object code.

**CSL-78-4 September 1978**

**USING ENCRYPTION FOR AUTHENTICATION IN LARGE NETWORKS OF COMPUTERS**
**by Roger M. Needham and Michael D. Schroeder**

Use of encryption to achieve authenticated communication in computer networks is discussed. Example protocols are presented for the establishment of authenticated connections, for the management of authenticated mail, and for signature verification and document integrity guarantee. Both conventional and public-key encryption algorithms are considered as the basis for protocols.

**CSL-78-5 September 1978**

**SEPARATING DATA FROM FUNCTION IN A DISTRIBUTED FILE SYSTEM**
**by Jay E. Israel, James G. Mitchell and Howard E. Sturgis**

This paper discusses an independent file facility, one that is *not* embedded in an operating system. The *distributed file system* (DFS) is so named because it is implemented on a cooperating set of *server* computers connected by a communications network, which together create the illusion of a single, logical system for the creation, deletion and random accessing of data. Access to the DFS can only be accomplished over the network; a computer (or, more precisely, a program running on one) that uses the DFS is

called a *client*. This paper describes the division of responsibility between servers and clients. We discuss examples of situations in which a client is expected to take prescribed steps in order to achieve its intended result. The basic tool for maintaining data consistency in these situations is the *atomic property* of file actions. This is a DFS feature that protects clients from system malfunctions and from the competing activities of other clients. We have implemented a experimental system based on these concepts.

**CSL-78-6 November 1978**

**CONSISTENT AND COMPLETE PROOF RULES FOR THE TOTAL CORRECTNESS OF PARALLEL PROGRAMS**
**by Lawrence Flon and Norihisa Suzuki**

We describe a formal theory of the total correctness of parallel programs, including such heretofore theoretically incomplete properties as safety from deadlock and starvation. We present a consistent and complete set of proof rules for the total correctness of parallel programs expressed in nondeterministic form.

The proof of consistency and completeness is novel in that we show that the weakest preconditions for the correctness criteria are actually fixed-points (least or greatest) of continuous functions over the complete lattice of total predicates. We have obtained proof rule schemata which can universally be applied to lead or greatest fixed points of continuous functions. Therefore, our proof rules are a priori consistent and complete once it is shown that certain weakest preconditions are extremum fixed-points. The relationship between true parallelism and nondeterminism is also discussed.

**CSL-79-1 January 1979**

## MONITORING SYSTEM BEHAVIOR IN A COMPLEX COMPUTATIONAL ENVIRONMENT

**by Mitchell L. Model**

Complex programming environments such as the representation systems constructed in Artificial Intelligence research present new kinds of difficulties for their users. A major part of program development involves debugging, but in a complex environment, the traditional tools and techniques available for this task are inadequate. Not only do traditional tools address state and process elements at too low a conceptual level, but an Artificial Intelligence System typically imposes its own data and control structures on top of those of its implementation language, thereby evading the reach of traditional program-level debugging tools. This work is directed at the development of appropriate monitoring tools for complex systems, in particular, the representation systems of Artificial Intelligence research.

The first half of this work provides the foundation for the design approach put forth and demonstrated in the second. Certain facts concerning limitations on human information processing abilities which formed the background for much of the research are introduced. The nature of computer programs is discussed, and a concept of "computational behavior" defined. A thematic survey of traditional debugging tools is presented, followed by a summary of recent work. Observation of program behavior ("monitoring") is shown to be the main function of most debugging tools and techniques. Concluding this first part is an analysis of the particular difficulties involved in monitoring the behavior of programs in large and complex AI systems.

The second half presents an approach to the design of monitoring facilities for complex systems. The need for system-level tools similar to the ones traditionally available is indicated. A new concept called "meta-monitoring" replaces traditional dumps and traces with selective reporting of high-level information *about* computations. The importance of the visually-oriented analogical presentation of high-level

information and the need to take into account differences between states and active processes are stressed. A generalized method for generating descriptions of system activity is developed. This method is based on a theoretical schematization of the fundamental structures and operations of computational systems and is easily instantiated for any particular AI system. Some specific display-based monitoring tools and techniques which were implemented for this work are exhibited. Several of the experimental monitoring facilities which were constructed in accordance with the principles of the proposed approach are described and their application to existing Artificial Intelligence Systems illustrated. While much of the research was performed in the context of the **KRL-1** system developed at Xerox Palo Alto Research Center, the general applicability of the theory and techniques of the present work is demonstrated by one of these facilities, which acts as a monitor for **MYCIN**, a medical diagnosis system developed at Stanford University that embodies knowledge in the form of production rules.

**CSL-79-3 April 1979**

**MESA LANGUAGE MANUAL VERSION 5.0**
**by J.G. Mitchell, W. Maybury, R. Sweet**

The Mesa language is one component of a programming system intended for developing and maintaining a wide range of systems and applications programs. Mesa supports the development of systems composed of separate modules with controlled sharing of information among them. The language includes facilities for user-defined data types, strong compile-time checking of both types and interfaces, procedure and coroutine control mechanisms, and control structures for dealing with concurrency and exceptional conditions.

**CSL-79-4 April 1979**

**TRANSPORT OF ELECTRONIC MESSAGES THROUGH A NETWORK**
**by R. Levin and M. Schroeder**

We list design objectives for a distributed mechanism to transport digital memoranda in a network, and discuss the associated administrative functions. We examine registering, authenticating, locating, and grouping users; define name mappings associated with message delivery; and consider the distribution of services among the computing elements in a network. Based on these analyses, we outline the structure for a distributed transport mechanism.

**CSL-79-5 June 1979**

**FORMALIZING THE ANALYSIS OF ALGORITHMS**
**by Lyle Harold Ramshaw**

Consider the average case analyses of particular deterministic algorithms. Typical arguments in this area can be divided into two phases. First, by using knowledge about what it means to execute a program, an analyst characterizes the probability distribution of the performance parameter of interest by means of some mathematical construct, often a recurrence relation. In the second phase, the solution of this recurrence is studied by purely mathematical techniques. Our goal is to build a formal system in which the first phases of these arguments can be reduced to symbol manipulation.

Formal systems currently exist in which one can reason about the correctness of programs by manipulating predicates that describe the state of the executing process. The construction and use of such systems belongs to the field of program verification. We want to extend the ideas of program verification, in particular, the partial correctness techniques of Floyd and Hoare, to allow assertions that describe the probabilistic state of the executing process to be written and manipulated. Ben Wegbreit proposed a system that extended Floyd-Hoare techniques to handle performance analyses, and we shall take Wegbreit's system as our starting point. Our

efforts at formal system construction will also lead us to a framework for program semantics in which programs are interpreted as linear functions between vector spaces of measures. This framework was recently developed by Dexter Kozen, and we shall draw upon his results as well.

We shall call our formal system the *frequency system*. The atomic assertions in this system specify the frequencies with which Floyd-Hoare predicates hold. These atomic assertions are combined with logical and arithmetic connectives to build assertions, and the rules of the frequency system describe how these assertions change as the result of executing program statements. The rules of the frequency system are sound, but not complete.

We then discuss the use of the frequency system in several average case analyses. In our examples, symbol manipulation in the frequency system leads directly to the recurrence relation that describes the distribution of the chosen performance parameter. The last of these examples is the algorithm that performs a straight insertion sort.

**CSL-79-6 June 1979**

**RASTER GRAPHICS FOR INTERACTIVE PROGRAMMING ENVIRONMENTS**
**by Robert F. Sproull**

Raster-scan display terminals can significantly improve the quality of interaction with conventional computer systems. The design of a graphics package to provide a "window" into the extensive programming environment of Interlisp is presented. Two aspects of the package are described: first, the functional view of display output and interactive input facilities as seen by the programmer, and second, the methods used to link the display terminal to the main computer via a packet-switched computer network. Recommendations are presented for designing operating systems and programming languages so as to simplify attaching display terminals. An appendix contains detailed documentation of the graphics package.

**CSL-79-7 June 1979**

## COMPACT ENCODINGS OF LIST STRUCTURE
**by Daniel G. Bobrow and Douglas W. Clark**

List structures provide a general mechanism for representing easily changed structured data, but can introduce inefficiencies in the use of space when fields of uniform size are used to contain pointers to data and to link the structure. Empirically determined regularity can be exploited to provide more space efficient encodings without losing the flexibility inherent in list structures. The basic scheme is to provide compact pointer fields big enough to accommodate most values that occur in them, and to provide "escape" mechanisms for exceptional cases. Several examples of encoding designs are presented and evaluated, including two designs currently used in Lisp machines. Alternative escape mechanisms are described, and various questions of cost and implementation are discussed. In order to extrapolate our results to larger systems than those measured, we propose a model for the generation of list pointers, and test the model against data from two programs. We show that according to our model, list structures with compact *cdr* fields will, as address space grows, continue to be compacted well with a fixed width small field. Our conclusion is that with a microcodable processor, about a factor of two gain in space efficiency for list structure can be had for little or no cost in processing time.

**CSL-79-8 October 1979**

## CODE GENERATION AND MACHINE DESCRIPTIONS
**by R.G.G. Cattell**

This is a collection of three papers covering a Ph.D. dissertation, "Formalization and Automatic Derivation of Code Generators," Cattell[1978]. The papers are revised reprints of versions appearing in the literature. The papers describe, respectively,

1. A code generator generator (this paper includes an overview of the thesis),

2. The model of machines (instruction set processors), and

3. The table-driven code generator and portions of the compiler in which it operates.

The tables for the code generator (3) are derived by the code generator generator (1) from the machine description (2). Thus the three papers describe the three main parts of the thesis work.

**CSL-79-9 November 1979**

**AN ENTITY-BASED DATABASE INTERFACE**
**by R.G.G. Cattell**

A user interface to a database designed for casual, interactive use is presented. The system is *entity-based*: the data display to the user is based upon entities (e.g., persons, documents, organizations) that participate in relationships, rather than upon relations alone as in the relational data model (Codd[1970]). Examples from an implementation of the system are shown, for a prototype personal database (PDB), developed in connection with the ZOG system at Carnegie-Mellon University (Robertson et al[1977]). Some details of the interface and associated issues concerning relational normal forms, views, and knowledge-based assistance are presented. Experience with the prototype system suggests that the entity-based presentation is appropriate for types of casual interactive use that existing database interfaces do not address, such as browsing. It is proposed that such an interface could be used to supplement a query language or other interface to allow users both kinds of views of the data.

**CSL-79-10 July 1979**

**PUP: AN INTERNETWORK ARCHITECTURE**
**by David R. Boggs, John F. Shoch, Edward A. Taft, and Robert M. Metcalfe**

Pup is the name of an internet packet format (*PARC Universal Packet*), a hierarchy of protocols, and a style of internetwork

communication. The fundamental abstraction is an end-to-end media-independent internetwork datagram. Higher levels of functionality are achieved by end-to-end protocols that are strictly a matter of agreement among the communicating end processes.

This report explores important design issues, sets forth principles that have guided the Pup design, discusses the present implementation in moderate detail, and summarizes experience with an operational internetwork. This work serves as the basis for a functioning internetwork system that provides service to about 1000 computers, on 25 networks of 5 different types, using 20 internetwork gateways.

**CSL-79-11 August 1979**

**ALTO: A PERSONAL COMPUTER**
**by C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull, and D.R. Boggs**

The Alto is a small computer system designed in early 1973 as an experiment in personal computing. Its principal characteristics, some of the design choices that led to the implementation, and some of the applications for which the Alto has been used are discussed.

**CSL-79-12 September 1979**

**VIOLET: AN EXPERIMENTAL DECENTRALIZED SYSTEM**
**by David Gifford**

Over the past year we have been designing and constructing an experimental decentralized information system called Violet. The lowest levels of the Violet system make it easy to construct a distributed user application by hiding the application's decentralized environment. Violet's first application, a calendar system, provides a sophisticated user interface to a simple relational data base manager. This paper presents our experience with the design and implementation of Violet. We discuss a new algorithm for replicated data which is implemented by Violet, and discoveries we have made about desirable concurrency modes for shared files. The

conclusion outlines what we consider to be desirable design features for decentralized information systems.

**CSL-79-13 October 1979**

**WFS: A SIMPLE SHARED FILE SYSTEM FOR A DISTRIBUTED ENVIRONMENT**
**by D. Swinehart, G. McDaniel and D. Boggs**

WFS is a shared file server available to a large network community. WFS responds to a carefully limited repertoire of commands transmitted through a network by client programs, and can be viewed as a remote intelligent disk controller. The system does not utilize network connections, but instead services independent page-level requests, one per per packet. The design emphasizes reliance upon client programs to implement the traditional facilities (stream I/0, a directory system, etc.) of a file system. The use of atomic file commands and connectionless network protocols nearly eliminates the need for WFS to maintain state information from request to request. Various uses of the system are discussed and extensions are proposed to provide security and protection without violating the design principles.

**CSL-79-14 September 1979**

**WEIGHTED VOTING FOR REPLICATED DATA**
**by David K. Gifford**

In a new algorithm for maintaining replicated data, every copy of a replicated file is assigned some number of votes. Every transaction collects a read quorum of $r$ votes to read a file, and a write quorum of $w$ votes to write a file, such that $r + w$ is greater than the total number of votes assigned to the file. This ensures that there is a non-null intersection between every read quorum and every write quorum. Version numbers make it possible to determine which copies are current. The reliability and performance characteristics of a replicated file can be controlled by appropriately choosing $r$, $w$, and the file's voting configuration. The algorithm guarantees serial consistency, admits temporary copies in a natural way by the introduction of copies with no votes, and has been

implemented in the context of an application system called Violet.

**CSL-80-1 January 1980**

**FORMAL SPECIFICATION AS A DESIGN TOOL**
**by J. Guttag and J. Horning**

The formulation and analysis of a design specification is almost always of more utility than the verification of the consistency of a program with its specification. Good specification tools can assist in this process, but have generally not been proposed and evaluated in this light. In this paper we outline a specification language combining algebraic axioms and predicate transformers, present part of a non-trivial example (the specification of a high-level interface to a display), and finally discuss the analysis of this specification.

**CSL-80-2 February 1980**

**THE ETHERNET LOCAL NETWORK: THREE REPORTS**
**by R. Metcalfe, D. Boggs, R. Crane, E. Taft, J. Shoch and J. Hupp**

This report reproduces previously published papers on the Ethernet local-area communications network:

- **Ethernet: Distributed Packet Switching for Local Computer Networks**, by Robert M. Metcalfe and David R. Boggs. Appeared in *Communications of the ACM*, vol. 19, no. 7, July 1976.

- **Practical Considerations in Ethernet Local Network Design**, by Ronald C. Crane and Edward A. Taft. Presented at the *Hawaii International Conference* on System Sciences, January 1980.

- **Measured Performance of an Ethernet Local Network**, by John F. Shoch and Jon A. Hupp. A preliminary version was presented at the *Local Area Communications Network Symposium*, Boston, May 1979.

The first paper introduces the Ethernet-style network, a multi-access broadcast packet-switched communication system, and presents the theory (CSMA/CD) that underlies it. The second paper discusses the design and implementation of the prototype Ethernet system, and the final paper presents performance results based on several years' practical experience with that system.

**CSL-80-3 March 1980**

**A CLIENT-BASED TRANSACTION SYSTEM TO MAINTAIN DATA INTEGRITY**
**by William H. Paxton**

This paper describes a technique for maintaining data integrity that can be implemented using capabilities typically found in existing file systems. Integrity is a property of a total collection of data. It cannot be maintained simply by using reliable primitives for reading and writing single units -- the relations between the units are important also. The technique suggested in this paper ensures that data integrity will not be lost as a result of simultaneous access or as a result of crashes at inopportune times. The approach is attractive because of its relative simplicity and its modest demands on the underlying file system. The paper gives a detailed description of how consistent, atomic transactions can be implemented by client processes communicating with one or more file server computers. The discussion covers file structure, basic client operations, crash recovery, and includes an informal correctness proof.

**CSL-80-4 March 1980**

**EFFICIENT DYNAMIC PROGRAMMING USING QUADRANGLE INEQUALITIES**
**by F. Frances Yao**

*Dynamic programming* is one of several widely used problem-solving techniques in computer science and operation research. In applying this technique, one always seeks to find speed-up by taking advantage of special properties of the problem at hand. However, in the current state of art, ad hoc

approaches for speeding up seem to be characteristic; few general criteria are known. In this paper we give a *quadrangle inequality* condition for rendering speed-up. This condition is easily checked, and can be applied to several apparently different problems. For example, it follows immediately from our general condition that the construction of optimal binary search trees may be speeded up from $O(n_3)$ steps to $O(n_2)$, a result that was first obtained by Knuth using a different and more complicated argument.

**CSL-80-5 December 1980**

**A LAYERED APPROACH TO SOFTWARE DESIGN**
**by Ira P. Goldstein and Daniel G. Bobrow**

Software engineers create alternative designs for their programs, develop these designs to various degrees, compare their properties, then choose among them. Yet most software environments do not allow alternative definitions of procedures to exist simultaneously. It is our hypothesis that an explicit representation for alternative designs can substantially improve a programmer's ability to develop software. To support this hypothesis, we have implemented an experimental Personal Information Environment (PIE) that has been employed to create alternative software designs, examine their properties, then choose one as the production version. PIE is based on the use of layered networks. Software systems are described in networks; alternatives are separated by being described in different layers. We also demonstrate that this approach has additional benefits as a data structure for supporting cooperative design among team members and as a basis for integrating the development of code with its associated documentation.

**CSL-80-6 October 1980**

**THE DISPLAY OF CHARACTERS USING GRAY LEVEL SAMPLE ARRAYS**
**by J.E. Warnock**

Character fonts on raster scanned display devices are usually represented by arrays of bits that are displayed as a matrix of

black and white dots. This paper reviews a filtering and sampling method as applied to characters for building multiple bit per pixel arrays. These arrays can be used as alternative character representations for use on devices with gray scale capability. Discussed in this paper are both the filtering algorithms that are used to generate gray scale fonts and some consequences of using gray levels for the representation of fonts including:

1. The apparent resolution of the display is increased when using gray scale fonts allowing smaller fonts to be used with higher apparent positional accuracy and readability. This is especially important when using low resolution displays.

2. Fonts of any size and orientation can be generated automatically from suitable high precision representations. This automatic generation removes the tedious process of "bit tuning" fonts for a given display.

**CSL-80-7 May 1980**

**DISPLAYED DATA STRUCTURES FOR INTERACTIVE DEBUGGING**
**by Brad A. Myers**

Many modern computer languages have a variety of basic data types and allow the programmer to define more. The facilities for debugging programs written in these languages, however, seldom provide any capabilities to capture the abstraction represented in the programmer's mind by the data types. *Incense*, the system described here, is a working prototype system that allows the programmer to interactively investigate data structures in programs. The desired displays can be specified by the programmer or a default can be used. The defaults include using the standard form for literals of the basic types, the actual names for enumerated types, stacked boxes for records, and curved lines with arrowheads for pointers. The intention is that the display produced should be similar to the picture the programmer would have drawn to explain the data type. Incense displays have the additional features that they can change dynamically.

Incense is written in and for the Pascal-like language *Mesa*, which was developed at the Xerox Palo Alto Research Center. Incense has been used to investigate and document many data structures including some of the internal data structures of the Incense system itself.

In addition to displaying data structures, Incense also allows the user to select, move, erase and redimension the resulting displays. Incense also allows the user to modify the actual values stored using the same high-level names that are displayed. These functions are provided in a uniform, natural manner using a pointing device ("mouse") and keyboard.

**CSL-80-8 June 1980**

**STRESS AND SALIENCE IN ENGLISH: THEORY AND PRACTICE**
**by Henry S. Thompson**

This work is concerned with various aspects of English prosody. On the practical side, a notational system and transcription methodology are proposed, and the results of an experiment to test their efficacy are reported. On the theoretical side, a formal model of that part of the speech production process which is concerned with prosody is presented.

Both the theoretical and practical sections of the work are founded on a relational, rhythmic approach to the manifestation of stress in English utterances. The transcription methodology is directed at notating the division of utterances into a succession of tone groups, each made up of rhythmic feet. Prosodic phenomena involving pitch are divided into two classes: tonal excursions and kinetic tones.

In an experiment to test this methodology, four subjects transcribed a 130 second monologue six times. Analysis of the results shows good agreement about the division into tone groups and feet, and about the location of kinetic tones, but less agreement about the nature of the kinetic tones, and about tonal excursions.

Using the consensus foot structure of the monologue which emerges from the experimental results, the hypothesis of the isochronic or stress-timed nature of English is investigated. No support for the hypothesis is found. Rather, a simple feature model based on whether a syllable is or is not first in a foot, and whether or not it is followed by a pause, is shown to provide a much better account of the duration of syllables.

A framework for a stage model of the speech production process, from concept to utterance, is proposed. The sub-stage of this model which divides an utterance into feet, called the footmaker, is described in detail. Variation of the possible divisions into feet of an utterance is attributed to various phonological, syntactic, semantic, and contextual factors. An inventory of features which encode these factors and of rules which operate in terms of these features is presented.

Finally, the model of the footmaker is applied to the consensus foot structure of the monologue, and the feature markings necessary to account for that structure are presented and discussed.

**CSL-80-9 June 1980**

**EFFICIENT ALGORITHMS FOR ENUMERATING INTERSECTING INTERVALS AND RECTANGLES**
**by Edward M. McCreight**

Let D be a dynamic set of intervals over the set 1,2..., $k$ of integers. Consider the following operations applied to D:

(1) Insert an interval [x,y] into D.
(2) Delete an interval [x,y] from D.
(3) Given a test integer u, list those intervals [x,y] in D that contain u.
(4) Given a test interval [u,v], list those intervals [x,y] in D for which [x,y] $\cap$ [u,v] is non-empty.

Using a new data structure that we call a tile tree, operations (1) and (2) can be implemented in $O(n \log(\max\{n,k\}))$ time, where $n$ is the cardinality of D. Operations (3) and (4) require

at most $O(n \log (\max\{n,k\})+s)$ time, where $s$ is the number of intervals listed. The tile tree occupies $O(n)$ space.

Using a previously-known plane-sweep technique, the operations above can be used to implement off-line algorithms to list all intersecting pairs of rectangles within a set of $n$ aligned rectangles in $O(n)$ space and $O(n \log n + s)$ time.

**CSL-80-10 June 1980**

**REQUIREMENTS FOR AN EXPERIMENTAL PROGRAMMING ENVIRONMENT**
**edited by L. Peter Deutsch and Edward A. Taft**

We define *experimental programming* to mean the production of moderate-size software systems that are usable by moderate numbers of people in order to test ideas about such systems. An *experimental programming environment* enables a small number of programmers to construct such experimental systems efficiently and cheaply -- an important goal in view of the rising cost of software.

In this report we present a catalog of programming environment capabilities and an evaluation of their cost, value, and relative priority. Following this we discuss these capabilities in the context of three existing programming environments: Lisp, Mesa, and Smalltalk. We consider the importance of specific capabilities in environments that already have them and the possibility of including them in environments that do not.

**CSL-80-11 October 1980**

**THE PROPER PLACE OF MEN AND MACHINES IN LANGUAGE TRANSLATION**
**by Martin Kay**

The only way in which the power of computers has been brought to bear on the problem of language translation is machine translation, that is, the automation of the entire process. Machine translation is an excellent research vehicle but stands no chance of filling actual needs for translation

which are growing at a great rate. In the quarter century during which work on machine translation has been going on, there has been considerable progress in relevant areas of computer science. However, advances in linguistics, important though they may have been, have not touched the core of this problem. The proper thing to do is therefore to adopt the kinds of solution that have proved successful in other domains, namely to develop cooperative man-machine systems. This paper proposes a *translator's amanuensis*, incorporating into a word processor some simple facilities peculiar to translation. Gradual enhancements of such a system could eventually lead to the original goal of machine translation.

**CSL-80-12 October 1980**

**ALGORITHM SCHEMATA AND DATA STRUCTURES IN SYNTACTIC PROCESSING**
**by Martin Kay**

The space in which models of human parsing strategy are to be sought is large. This paper is an exploration of that space, attempting to show what its dimensions are and what some of the choices are that the psycholinguist must face. Such an exploration as this may provide some protection against the common tendency to let some choices go by default.

A notion of configuration tables is used to locate algorithms on two dimensions according as (1) they work top-down or bottom-up, and (2) they are directed or undirected. The algorithms occupying a particular place in this two dimensional space constitute an algorithm schema. The notion of a chart is used to show how to limit the amount of work a parser must do by ensuring that nothing is done more than once. Finally, the notion of an agenda is introduced to show how a rich variety of psychological strategies can be combined in a principled way with a given algorithm schema to yield an algorithm.

**CSL-81-1 January 1981**

**THE DORADO: A HIGH-PERFORMANCE PERSONAL COMPUTER**
**THREE PAPERS**
**by B. Lampson, K. Pier, G. McDaniel, S. Ornstein, and D. Clark**

This report reproduces three papers on the Dorado personal computer. Each has been, or will be, published in a journal or proceedings.

**A Processor for a High-Performance Personal Computer**, by Butler W. Lampson and Kenneth A. Pier. Appeared in the Proceedings of the 7th Symposium on Computer Architecture, SigArch/IEEE, La Baule, May 1980, pp. 146-160.

**An Instruction Fetch Unit for a High-Performance Personal Computer**, by Butler W. Lampson, Gene McDaniel, and Severo M. Ornstein. Appeared in IEEE Transactions on Computers, vol. C-33, no. 8, August 1984, pp. 712-730.

**The Memory System of a High-Performance Personal Computer**, by Douglas W. Clark, Butler W. Lampson, and Kenneth A. Pier. A revised version appeared in IEEE Transactions on Computers, vol. C-30, no. 10, October 1981.

The first paper describes the Dorado's micro-programmed processor, and also gives an overview of its history and physical construction. The second discusses the instruction fetch unit which prepares program instructions for execution, and the third deals with the cache, map, and main storage of the Dorado's memory system.

**CSL-81-2 January 1981**

**THE TXDT PACKAGE -- INTERLISP TEXT EDITING PRIMITIVES**
**by J. Strother Moore**

The TXDT package is a collection of INTERLISP programs designed for those who wish to build text editors in INTERLISP. TXDT provides a new INTERLISP data type, called a buffer, and programs for efficiently inserting, deleting, searching and manipulating text in buffers. Modifications may be made undoable. A unique feature of TXDT is that an address may be "stuck" to a character occurrence so as to follow that character wherever it is subsequently moved. TXDT also has provisions for fonts.

**CSL-81-3 March 1981**

**AN EXPERIMENTAL DESCRIPTION-BASED PROGRAMMING ENVIRONMENT: FOUR REPORTS**
**by Ira Goldstein and Daniel Bobrow**

This document reprints four articles that describe PIE, an experimental personal information environment, from the vantage point of its application to software development. PIE employs a description language to support the interactive development of programs. PIE contains a network of nodes, each of which can be assigned several perspectives. Each perspective describes a different aspect of the program structure represented by the node, and provides specialized actions from that point of view. Contracts can be created that monitor nodes describing different parts of a program's description. Contractual agreements are expressible as formal constraints, or, to make the system failsoft, as English text interpretable by the user. Contexts and layers are used to represent alternative designs for programs described in the network. The layered network database also facilitates cooperative program design by a group, and coordinated, structured documentation.

The first article, "Descriptions for a Programming Environment," provides an overview of PIE. The second article, "Extending Object Oriented Programming in Smalltalk,"

explores the generalizations made to the Smalltalk language in order to combine its strengths as an object language with capabilities usually found in AI description languages. This extended dialect is used to implement the PIE system. The third article, "Representing Design Alternatives," describes PIE's machinery for representing the evolution of a software design. This machinery is described in greater detail in a separate report, CSL-80-5. The fourth article, "Browsing in a Programming Environment," describes the user interface.

**CSL-81-5 January 1982**

**PRIORITY SEARCH TREES**
**by Edward M. McCreight**

Let D be a dynamic set of ordered pairs [x,y] over the set 0,1,...,*k-1* of integers. Consider the following operations applied to D:

(1) Insert (delete) a pair [x,y] into (from) D.

(2) Given test integers x0, x1, and y1. among all pairs [x,y] in D such that $x0 \le x \le x1$ and $y \le y1$, find a pair whose x is minimal (maximal).

(3) Given test integers x0 and x1, among all pairs [x,y] in D such that $x0 \le x \le x1$, find a pair whose y is minimal.

(4) Given test integers x0, x1. and y1, enumerate those pairs [x,y] in D such that $x0 \le x \le x1$ and $y \le y1$.

Using a new data structure that we call a priority search tree, of which two variants are introduced, operations (1), (2), and (3) can be implemented in O(log *n*) time, where *n* is the cardinality of D. Operation (4) requires at most O(log $n+s$) time, where *s* is the number of pairs enumerated. The priority search tree occupies O(*n*) space.

Priority search tree algorithms can be used effectively as subroutines in diverse applications. With them one can answer questions of intersection or containment in a dynamic set of linear intervals. Using a previously-known plane-sweep

technique, they can be used to implement off-line algorithms to enumerate all intersecting pairs of rectangles. Priority search trees can also be used to implement best-/first-fit storage allocation.

**CSL-81-6 August 1981**

**LAUREL MANUAL**
**by Douglas K. Brotz**

Laurel is an Alto-based, display-oriented, computer mail system interface. It provides facilities to retrieve mail and present it for delivery, and to display, forward, classify, file, edit and print messages. Additional features include facilities to read, write and copy files, run programs, and a whole lot more. Laurel is a component of a distributed message system that has been in operation for several years in the Xerox Research Internet.

This document is a description of the facilities contained in Laurel. Several tips on proper use of computer mail facilities in a social context are included.

**CSL-81-7 June 1981**

**TRELLIS DATA COMPRESSION**
**by Lawrence Colm Stewart**

Tree and trellis data compression systems have traditionally been designed by using a tree or trellis search algorithm to improve the performance of traditional coding systems such as adaptive delta modulation or predictive quantization. Recent work in the area of vector quantization has suggested the possibility of designing new tree and trellis codes which are well matched to particular sources. The main design procedure iterates on a long training sequence to improve the performance of an initial trellis decoder. An additional procedure, given a trellis decoder, can produce a decoder of longer constraint length which performs at least as well. Combined, these algorithms provide a complete design procedure for trellis encoding data compression systems.

For random sources, many existing data compression systems can be readily improved and performance close to the rate-distortion bound can be obtained. In the applications area of speech compression, tree and trellis codes designed with these algorithms permit the construction of low rate speech waveform coders, low rate residual excited linear predictive coders (RELP), and a new kind of hybrid tree coder which provides good quality speech at rates below 7000 bits per second.

**CSL-81-8 June 1981; Revised March 1982**

**INFORMATION STORAGE IN A DECENTRALIZED COMPUTER SYSTEM**
**by David K. Gifford**

This paper describes an architecture for shared information storage in a decentralized computer system. The issues that are addressed include: naming of files and other objects (naming), reliable storage of data (stable storage), coordinated access to shared storage (transactional storage), location of objects (location), use of multiple copies to increase performance, reliability and availability (replication), dynamic modification of object representations (reconfiguration), and storage security and authentication (protection).

A complete model of the architecture is presented, which describes the interface to the facilities provided, and describes in detail the proposed mechanisms for implementing them. The model presents new approaches to naming, location, replication, reconfiguration, and protection. To verify the model, three prototypes were constructed, and experience with these prototypes is discussed.

The model names objects with variable length byte arrays called references. References may contain location information, protection guards, cryptographic keys, and other references. In addition, references can be made indirect to delay their binding to a specific object or location.

The replication mechanism is based on assigning votes to each copy of a replicated object. The characteristics of a replicated

object can be chosen from a range of possibilities by appropriately choosing its voting configuration. Temporary copies can be easily implemented by introducing copies with no votes.

The reconfiguration mechanism allows the storage that is used to implement an object to change while the system is operating. A client need not be aware that an object has been reconfigured.

The protection mechanism is based on the idea of sealing an object with a key. Sealed objects can only be unsealed with an appropriate set of keys. Complex protection structures can be created by using such operators as Key-Or and Key-And. The protection mechanism can be employed to create popular protection mechanisms such as capabilities, access control lists, and information flow control.

**CSL-81-9 May 1981**

**REMOTE PROCEDURE CALL**
**by Bruce Jay Nelson**

Remote procedure call is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel. The thesis of this dissertation is that remote procedure call (RPC) is a satisfactory and efficient programming language primitive for constructing distributed systems.

A survey of existing remote procedure mechanisms shows that past RPC efforts are weak in addressing the five crucial issues: uniform call semantics, binding and configuration, strong typechecking, parameter functionality, and concurrency and exception control. The body of the dissertation elaborates these issues and defines a set of corresponding *essential properties* for RPC mechanisms. These properties must be satisfied by any RPC mechanism that is fully and uniformly integrated into a programming language for a homogeneous distributed system. Uniform integration is necessary to meet the dissertation's fundamental goal of syntactic and semantic

*transparency* for local and remote procedures. Transparency is important so that programmers need not concern themselves with the physical distribution of their programs.

In addition to these essential language properties, a number of *pleasant properties* are introduced that ease the work of distributed programming. These pleasant properties are good performance, sound remote interface design, atomic transactions, respect for autonomy, type translation, and remote debugging.

With the essential and pleasant properties broadly explored, the detailed design of an RPC mechanism that satisfies all of the essential properties and the performance property is presented. Two design approaches are used: The first assumes full programming language support and involves changes to the language's compiler and binder. The second involves no language changes, but uses a separate translator -- a source-to-source RPC compiler -- to implement the same functionality.

Design decisions crucial to the efficiency of the mechanism are made using a set of RPC performance lessons. These lessons are based on the empirical performance evaluation of a sequence of five working RPC mechanisms, each one faster than its predecessor. Some expected results about the costs of parameter copying, process switching, and runtime type manipulation are confirmed; a surprising result about the price of protocol layering is presented as well. These performance lessons, applied in concert, reduce the roundtrip time for a remote procedure call by a remarkable factor of 35. For moderate speed personal computers communicating over an Ethernet, for example, a simple remote call takes 800 microseconds: on a higher speed personal computer, the same remote call takes 149 microseconds. In both cases the remote call takes about 20 times longer than the same local call. This represents a substantial performance improvement over other operational RPC mechanisms.

**CSL-81-10 June 1981**

**TECHNIQUES FOR PROGRAM VERIFICATION**
**by Greg Nelson**

The main subject of this paper is the detailed description of a mechanical theorem prover for use in program verification, following the approach used successfully in the simplifier/theorem-prover of the Stanford Pascal Verifier: algorithms are developed for theorem-proving in various logical theories, then the algorithms are combined to produce a theorem-prover for the "combination" of the theories. The algorithms described in this paper are refinements of those used in coding the Stanford Verifier.

More precisely, algorithms are described for determining the satisfiability of conjunctions of literals (i.e., atomic formulas or negations thereof) in the following theories: the theory of the real numbers under $+$, $-$, $<$, and $\leq$; the theory of Lisp list structure under **car**, **cdr**, **cons**, **atom**, and $=$; the theory of uninterpreted function symbols under $=$, and the theory of arrays under operations for accessing and updating elements. A general method for combining such algorithms is described and applied to them to produce a single program that determines the satisfiability of conjunctions of literals containing any of the above functions and predicates.

A second subject of the paper is the problem of combining these "special-purpose" theorem-proving techniques with the two "general-purpose" techniques of matching (or resolution) and induction. Several examples are worked through that suggest that matching in the data structure of the special-purpose algorithms is a viable alternative to matching using ordinary list structure, but the details of an appropriate matcher are not considered. Inductive proofs are considered in more detail: formal semantics are described for a logical system that extends first-order logic with non-deterministic partial recursive function definitions, and an appropriate induction rule is defined and proved to be valid. The rule seems to be suited to the methods of Boyer and Moore; if so, their successful heuristics for proving theorems about total functions can be used with the new rule to prove theorems

about partial functions. This is an important extension, since the specifications for programs that manipulate linked data structures are most naturally written using partial (and perhaps non-deterministic) recursive function definitions. An example is given of the use of the system in verifying the correctness of a "destructive" list reversal program.

Finally, this paper addresses the question of the role of program verification in contemporary programming environments. It is argued that there are immediate applications for program verification in establishing invariants that are of practical value to a compiler. A programming language is outlined that uses verifiable invariants in place of type declarations, thereby allowing both a higher level of consistency checking than that performed by compilers for conventional hard-typed languages, and also the flexible data structures that are ruled out by strict type systems.

**CSL-81-11 July 1981**

**REAL PROGRAMMING IN FUNCTIONAL LANGUAGES**
**by James H. Morris**

The established properties of functional languages -- easy to define semantics and mathematical elegance -- are appealing to meta-programmers who study programming and programs at one remove. Most people believe that functional programming is inappropriate for real programmers who write programs that are judged on their behavior rather than their appearance. We shall explore this question by considering experience with two languages, Poplar and Euclid, that have a claim to being functional languages and to being used on real problems -- string processing and system programming, respectively.

**CSL-81-12 October 1981**

**REPORT ON THE PROGRAMMING LANGUAGE EUCLID**
**by Butler Lampson, James Horning, Ralph London, James Mitchell and Gerald Popek**

This report describes a programming language called Euclid, intended for the expression of system programs which are to be verified. Euclid draws heavily on Pascal for its structure and many of its features. In order to reflect this relationship as clearly as possible, the Euclid report has been written as a heavily edited version of the revised Pascal report.

Proof rules for Euclid appear in a separate report [London et al. 1978]. An informal discussion of the language design appears in [Popek et al. 1977]. Euclid has been implemented (with some omissions) by the Computer Systems Research Group, University of Toronto, Toronto, Canada, and I.P. Sharp Associates, Toronto, Canada [Holt et al. 1978, Holt and Wortman 1979, Holt et. al. 1980]. The translator is a 70,000 line Euclid program, the largest such program now in existence [Wortman and Cordy 1981].

This is the fourth version of the Euclid report; earlier versions appeared in May 1976, August 1976, and February 1977 (the latter as SIGPLAN Notices 12, 2, Feb. 1977).

**CSL-82-1 February 1982**

**CRYPTOGRAPHIC SEALING FOR INFORMATION SECRECY AND AUTHENTICATION**
**by David K. Gifford**

A new protection mechanism is described that provides general primitives for protection and authentication. The mechanism is based on the idea of sealing an object with a key. Sealed objects are self-authenticating, and in the absence of an appropriate set of keys, only provide information about the size of their contents. New keys can be freely created at any time, and keys can also be derived from existing keys with operators that include Key-And and Key-Or. This flexibility allows the protection mechanism to implement common

protection mechanisms such as capabilities, access control lists, and information flow control. The mechanism is enforced with a synthesis of conventional cryptography, public-key cryptography, and a threshold scheme.

**CSL-82-2 May 1982**

**AN ANALYSIS OF A MESA INSTRUCTION SET**
**by Gene McDaniel**

This paper reports measurements of dynamic instruction frequencies for two Mesa programs running on a Dorado personal computer at the Computer Science Laboratory of the Xerox Palo Alto Research Center. The patterns of use associated with the Mesa instruction set are examined in order to find the implications of that usage for the Mesa architecture and its implementation. This paper discusses Mesa's byte encoding, patterns of memory references, use of an expression evaluation stack, and the costs of emulating 32-bit operations on a 16-bit processor.

**CSL-82-3 June 1982**

**SOME NOTES ON PUTTING FORMAL SPECIFICATIONS TO PRODUCTIVE USE**
**by John Guttag*, Jim Horning, and Jeannette Wing***

These notes are personal reflections, stemming from attempts to understand the sources of problems and successes in the application of work on formal specifications. Our intent is to provoke thought about the nature and value of work in the area; not to provide a set of well-tested results. Rather than focusing on yet another specification language, we have tried to take a broad view of the role of formal specifications in the program development process.

*MIT Laboratory for Computer Science

**CSL-82-4 July 1982**

**GRAPEVINE: AN EXERCISE IN DISTRIBUTED COMPUTING**
**by Andrew Birrell, Roy Levin, Roger Needham*, Michael Schroeder**

Grapevine is a multicomputer system on the Xerox research internet. It provides facilities for the delivery of digital messages such as computer mail; for naming people, machines, and services; for authenticating people and machines; and for locating services on the internet. This paper has two goals: to describe the system itself and to serve as a case study of a real application of distributed computing. Part I describes the set of services provided by Grapevine and how its data and function are divided among computers on the internet. Part II presents in more detail selected aspects of Grapevine that illustrate novel facilities or implementation techniques, or that provide insight into the structure of a distributed system. Part III summarizes the current state of the system and the lessons learned from it so far.

*University of Cambridge Computer Laboratory

**CSL-82-5 March 1982**

**PACKET-VOICE COMMUNICATIONS ON AN ETHERNET LOCAL COMPUTER NETWORK: AN EXPERIMENTAL STUDY**
**by Timothy A. Gonsalves**

Local computer networks have been used successfully for data applications such as file transfers for several years. Recently, there have been several proposals for using these networks for voice applications. This paper describes a simple voice protocol for use on a packet-switching local network. This protocol is used in an experimental study of the feasibility of using a 3 Mbps experimental Ethernet network for packet-voice communications. The study shows that with appropriately chosen parameters the experimental Ethernet is capable of supporting about 40 simultaneous 64-Kbps voice conversations with acceptable quality. This corresponds to a utilization of 95% of the network capacity.

**CSL-82-7 December 1982**

## CONTROLLING LARGE SOFTWARE DEVELOPMENT IN A DISTRIBUTED ENVIRONMENT

**by Eric Emerson Schmidt**

Breaking a program up into modules is an important technique for managing the complexity of large systems. As the number of modules increases, the modules themselves need to be managed. Changing even a single module can be difficult. Compilation and loading are complicated. Saving the state of a program for others to build on is quite error-prone. The development of a large program as part of a multi-person project is even worse. This thesis presents solutions to these problems. We use new languages to describe the modules that comprise a system and tools that automate software development. The first solution developed is a version control system of modest goals that has been used to maintain up to 450,000 lines of code over the past year. Users of this system list versions of files in *description files* (DF files) that are automatically maintained for the user. DF files may refer to other DF files when one software package depends on another. A working set of software that is saved in a safe location is called a release. The need for a *release process* was identified and an iterative algorithm that uses DF files to perform releases has been developed. Based on experience with the DF system and the desire to automate the entire compile-edit-debug-release cycle, a second solution was developed in which the development cycle is controlled by the *System Modeller*. The modeller automatically manages the compilation, loading, and saving of new modules as they are produced. The user describes his software in a *system model* that lists the versions of files used, the information needed to compile the system, and the interconnections between the various modules. The modeller is connected to the editor and is notified when files are edited and new versions are created. To provide fast response, the modeller behaves like an incremental compiler: only those modules that change are analyzed and recompiled.

**CSL-83-1 January 1983**

## AN INTERACTIVE HIGH-LEVEL DEBUGGER FOR CONTROL-FLOW OPTIMIZED PROGRAMS

**by Polle T. Zellweger**

The transformations performed by an optimizing compiler have traditionally impeded interactive debugging in source language terms. A prototype system, called Navigator, has been developed for debugging optimized programs written in Cedar, an Algol-like language. Navigator can be used to monitor program execution flow in the presence of two optimizations: inline procedure expansion and cross-jumping (merging identical tails of code paths that join). This paper describes the problems that these two optimizations create for debugging and Navigator's solutions to these problems. The approach taken is to collect extra information during the optimization phases of compilation. At runtime, Navigator uses the additional information to hide the effects of the optimizations from the programmer.

**CSL-83-2 January 1983**

## MOCKINGBIRD: A COMPOSER'S AMANUENSIS

**by John T. Maxwell III and Severo M. Ornstein**

Mockingbird is a computer based, display oriented, music notation editor. It is especially focused on helping a composer to capture his ideas. It can play scores on the synthesizer as well as display and print them in standard notational form. It can accept both graphical input and input played on a synthesizer keyboard attached to the computer. In the latter case, the user must edit the music to turn it into standard notational form, and much of Mockingbirds interest lies in the methods by which this conversion is accomplished. The editor is highly interactive, presenting the illusion that the user can reach in and move elements of the score around as desired. This illusion is achieved by always showing the detail of the score exactly as it might be printed.

**CSL-83-3 October 1983**

**INTERNET BROADCASTING**
**by David R. Boggs**

Broadcasting should be a standard addressing mode of all packet-switched computer networks. Further, when networks are interconnected to form an internet, a broadcast mechanism is also required.

Broadcasting is the delivery of a packet to all hosts in a network; unicasting is the delivery of a packet to one specific host. They are distinct forms of interprocess communication; functions that are simple to do with one are difficult to do using only the other.

A broadcast is used when you don't know whom specifically to address. There are two situations where this occurs: 1) when you are searching for some information but you don't know whom to ask (for example, standing up in a theatre and saying "Is there a doctor in the house?"), and 2) when you possess some information of use to others but you don't know specifically whom (for example, standing up in a theatre and yelling, "Fire!").

A network should give its best efforts to deliver a copy of a broadcast packet to each host, but perfectly reliable delivery is not required. It is sufficient that most hosts receive a broadcast and that the same hosts not miss retransmissions. Just as with unicasting, higher-level protocols can be used to improve the reliability of the basic broadcast delivery mechanism, if required. Such an "unreliable" broadcast mechanism is straightforward to implement in all types of packet-switched networks.

In an internet composed of possibly thousands of networks and millions of hosts, a full internet-wide broadcast, the obvious internet analog to broadcasting in a single network, is seldom the right choice. A directed broadcast, delivery of a packet to all hosts on any single network in an internet, is simpler to implement, closer to what most users need, and sufficient to construct many forms of broadcast-based

interprocess communication, including an internet-wide broadcast.

**CSL-83-4 May 1983**

**DESIGN AND IMPLEMENTATION OF A RELATIONSHIP-ENTITY-DATUM DATA MODEL**
**by R.G.G. Cattell**

The Model level of the Cypress Database Management System, built upon the earlier Cedar DBMS, provides data description and access capabilities at a higher level of abstraction than the existing system and other conventional DBMS's. In this report we describe the design of the Cypress data model and discuss issues in the efficient implementation of such a model. Cypress incorporates features motivated by experience with local database applications. It may be viewed as an integration of a number of existing data models; we present the criteria that led to this choice. The Cypress primitives include simple data values such as strings or integers, entities representing real or abstract objects, and relationships among entities and/or simple data values. We also provide mechanisms for a hierarchy of types, relational keys, and segmentation of databases into independent files. Cypress allows a conventional relational query language. We argue that our extensions to simpler data models allow a more powerful and efficient implementation, and we describe the optimizations Cypress performs. We also discuss some preliminary experience with user tools and applications developed in conjunction with Cypress.

**CSL-83-5 March 1984**

**DATA TYPES ARE VALUES**
**by James Donahue and Alan Demers***

An important goal of programming language research is to isolate the fundamental concepts of languages, those basic ideas that allow us to understand the relationship among various language features. This paper examines one of these underlying notions, data type, with particular attention to the

treatment of generic or polymorphic procedures and static type-checking.

*Cornell University Computer Science Department

**CSL-83-6 December 1983**

**PRELIMINARY REPORT ON THE LARCH SHARED LANGUAGE**
**by J.V. Guttag* and J.J. Horning**

Each member of the Larch family of formal specification languages has a component derived from a programming language and another component common to all programming languages. We call the former interface languages, and the latter the Larch Shared Language.

This report presents version 1.0 of the Larch Shared Language. It begins with a brief introduction to the Larch Project and the Larch family of languages. The next chapter presents most of the features of the Larch Shared Language and briefly discusses how we expect these features to be used. It should be read before reading either of the remaining two chapters, which are a self-contained reference manual and a set of examples.

*MIT Laboratory for Computer Science

**CSL-83-7 December 1983**

**IMPLEMENTING REMOTE PROCEDURE CALLS**
**by Andrew D. Birrell and Bruce Jay Nelson**

Remote procedure calls (*RPC*) appear to be a useful paradigm for providing communication across a network between programs written in a high level language. This paper describes a package providing a remote procedure call facility, the options that face a designer of such a package, and the decisions we made. We describe the overall structure of our RPC mechanism, our facilities for binding RPC clients, the transport level communication protocol, and some performance measurements. We include descriptions of some

optimizations we used to achieve high performance and to minimize the load on server machines that have many clients.

**CSL-83-8 February 1984**

**ADDING VOICE TO AN OFFICE COMPUTER NETWORK**
**by D.C. Swinehart, L.C. Stewart and S.M. Ornstein**

This paper describes the architecture and initial implementation of an experimental telephone system developed by the Computer Science Laboratory at the Xerox Palo Alto Research Center (PARC CSL). A specially designed processor (Etherphone™) connects to a telephone instrument and transmits digitized voice, signaling, and supervisory information in discrete packets over the Ethernet local area network. When used by itself, an Etherphone processor provides the functions of a conventional telephone, but it comes into its own when combined with the capabilities of a nearby office workstation, a voice file service, and other shared services such as databases. Most of the work so far has gone into the basic provisions for voice switching and transmission. Today the system supports ordinary telephone calls and simple voice message services. We will expand these functions as we explore the integration of voice with our experimental office systems.

**CSL-83-9 April 1984**

**THE SEMANTICS OF LAZY (AND INDUSTRIOUS) EVALUATION**
**by Robert Cartwright* and James Donahue**

Lazy evaluation has gained widespread acceptance among language theoreticians -- particularly among the advocates of "functional programming." The implementation of lazy evaluation is easy to describe, but its semantic consequences are deceptively complex. This paper develops a comprehensive semantic theory of lazy evaluation as a change in the value space over which computation is performed. It also explores several approaches to formalizing the theory of lazy evaluation within a programming logic.

*Rice University

**CSL-83-10 November 1983**

**DEFTLY REPLACING go to STATEMENTS WITH exit's**
**by Lyle Ramshaw**

Suppose that we are trying to eliminate the **go to** statements in a PASCAL program by replacing them with block **exit** statements; and suppose that we aren't willing either to introduce new variables or to replicate code. Previous research has shown that, under this policy, we can eliminate all of the **go to's** from a program if and only if that program's flow graph is reducible. In this paper, we shall investigate the extent to which **go to's** can be eliminated under still stricter policies. First, we shall discuss two simple program transformations that replace **go to's** with equivalent **exit's** simply by adding new levels of block structure. We shall find that these transformations by themselves are sufficient to eliminate all of the **go to's** in a program if and only if that program has no head-to-head pairs of **go to's**, where we call a pair of **go to's** *head-to-head* if each jumps into the other's interior. Second, we shall consider an intermediate policy in which we don't restrict ourselves to the two transformations but we do forbid any reordering of the atomic tests and actions of the program. Under this intermediate policy, we shall find that all **go to's** can be eliminated if and only if a certain augmented version of the flow graph is reducible. The theory of these augmented flow graphs elucidates the relationship between the standard theory of flow graph reducibility and our results concerning head-to-head pairs of **go to's**.

**CSL-83-11 June 1984**

**THE CEDAR PROGRAMMING ENVIRONMENT: A MIDTERM REPORT AND EXAMINATION**
**by Warren Teitelman**

This collection of papers comprises a report on Cedar, a state-of-the-art programming system. Cedar combines in a single integrated environment: high-quality graphics, a sophisticated

editor and document preparation facility, and a variety of tools for the programmer to use in the construction and debugging of his programs. The Cedar Programming Language is a strongly-typed, compiler-oriented language of the Pascal family. What is especially interesting about the Cedar project is that it is one of the few examples where an interactive, experimental programming environment has been built for this kind of language. In the past, such environments have been confined to dynamically typed languages like Lisp and Smalltalk.

The first paper, "The Roots of Cedar," describes the conditions in 1978 in the Xerox Palo Alto Research Center's Computer Science Laboratory that led us to embark on the Cedar project and helped to define its objectives and goals. Important decisions had to be made about what facilities and features were essential versus simply desirable, both with regard to the programming language as well as tools and packages. This section not only presents these decisions, but also describes the process by which we reached them. These deliberations are especially interesting in light of the fact that three communities with diverse programming languages (Mesa, Lisp, and Smalltalk) and very different programming styles, met to discuss the merits and drawbacks of their individual systems and religions, with the purpose of reaching some sort of consensus that would allow the construction of a programming environment that would be satisfactory to all of them.

The second paper, "A Tour Through Cedar," is essentially a travelogue through the current Cedar environment (as of September 1983) in the form of a transcript of an actual session. This transcript consists of numerous snapshots of the display screen interspersed with dialogue and commentary. The intent is to produce an effect similar to that of the reader sitting down with a user in front of a display terminal and being given a live demonstration of the system, while an expert comments on some of the why's and wherefore's. During the course of this demonstration, the reader is introduced to most of the salient features of the Cedar Programming Environment as they come up and are used. In many cases we will digress from this demonstration to discuss

some aspect of these features, such as why we did it this way, how important this particular facility actually turned out to be, etc. The final paper, "Cedar: The Report Card," discusses and attempts to evaluate how well we have succeeded in reaching our objectives and goals, to what extent the original objectives and goals were changed or evolved during the course of the project, and what remains to be done.

**CSL-83-12 December 1983**

**GRAPEVINE: TWO PAPERS AND A REPORT**
**by Andrew Birrell, Roy Levin, Roger Needham, and Michael Schroeder**

Grapevine is a multi-computer system on the Xerox research internet. It provides facilities for the delivery of digital messages such as computer mail; for naming people, machines and services; for authenticating people and machines; and for locating services on the internet. This report reproduces an earlier paper on Grapevine (CSL-82-4, "Grapevine: An Exercise in Distributed Computing"), a paper on our experience with use of the Grapevine system, and a detailed description of the services provided by Grapevine.

**CSL-83-15 December 1983**

**A DESCRIPTION OF THE CEDAR LANGUAGE**
**A CEDAR LANGUAGE REFERENCE MANUAL**
**by Butler W. Lampson**

The Cedar Language is a programming language derived from Mesa, which in turn is derived from Pascal. It is meant to be used for a wide variety of programming tasks, ranging from low-level system software to large applications. In addition to the sequential control constructs, static type checking and structured types of Pascal, and the modules, exception handling, and concurrency control constructs of Mesa, Cedar also has garbage collection, dynamic types, and a limited form of type parameterization.

This report describes the Cedar language. Except for Chapter 2, it is written strictly in the style of a reference manual, not a

tutorial. Furthermore, it describes the entire language, including a number of obsolete constructs and historical accidents. Hence it tells much more than you probably want to know. A summary of the safe language and comments throughout the manual suggest which constructs should be preferred for new programs.

**CSL-84-4 October 1984**

**THE ALPINE FILE SYSTEM**
**by Mark R. Brown, Karen Kolling, and Edward A. Taft**

Alpine is a file system that supports atomic transactions and is designed to operate as a service on a Computer internet. Alpine's primary purpose is to store files that represent databases; an important secondary goal is to store ordinary files representing documents, program modules, and the like.

Unlike other file servers described in the literature, Alpine uses a log-based technique to implement atomic file update. Another unusual aspect of Alpine is that it performs all communication via a general-purpose remote procedure call facility. Both of these decisions have worked out well. This paper describes Alpine's design and implementation, and evaluates the system in light of our experience to date.

Alpine is written in Cedar, a strongly typed modular programming language that includes garbage-collected storage. This paper reports on using the Cedar language and programming environment to develop Alpine.

**CSL-84-5 May 1984**

**INTERACTIVE SOURCE-LEVEL DEBUGGING OF OPTIMIZED PROGRAMS**
**by Polle Trescott Zellweger**

The transformations performed by an optimizing compiler have traditionally impeded interactive debugging in source language terms: after optimization, a program's source text and object code do not have a straightforward correspondence. This dissertation shows that effective interactive source-level

debuggers can be provided for optimized programs. Such debuggers can reduce debugging time and programmer confusion. These benefits are especially important given the increasing availability of optimizing compilers.

The first half of the dissertation studies the overall problem of debugging optimized programs. It presents a general view of debuggers and defines two important levels of debugger behavior for optimized programs. A debugger provides *expected behavior* if it hides the effects of the optimizations from the user by doing behind-the-scenes processing. It provides *truthful behavior* if it indicates that it cannot give the exact answer to a debugging query (because the executing program differs from the source program). The user may be able to deduce the correct answer from the partial information displayed by a truthful response. A thorough study of the interactions between optimization and debugging is included. In addition, a collection of solution techniques to relieve the problems caused by optimization are described.

The second half of the dissertation describes implementation experience with one aspect of the problem. A prototype debugging system called Navigator was developed for the Cedar programming environment at the Xerox Palo Alto Research Center. Navigator can be used interactively to monitor program execution flow in the presence of two simple but nontrivial optimizations inline procedure expansion and cross-jumping (merging identical tails of code paths that join). Navigator provides expected behavior by combining information collected by the compiler about the effects of the optimizations and information collected by the debugger about the control-flow history of the computation. Program execution space and speed are almost totally unaffected when no debugging requests are active. When debugging is requested, Navigator provides its added functionality without noticeably degrading debugger response time for most programs. Proofs of correctness of the compiler and debugger algorithms are given, as well as an analysis of their efficiency.

**CSL-84-6 July 1985**

**EXPERIENCE WITH THE CEDAR PROGRAMMING ENVIRONMENT FOR COMPUTER GRAPHICS RESEARCH**
**by Richard J. Beach**

Cedar is an integrated programming environment for building experimental Computer systems. The environment consists of a well-coordinated collection of tools and packages and a language that encourages and enforces their coordination. Cedar incorporates a device-independent imaging model for presenting all information and relies on input interaction techniques to control the environment.

The computer graphics research accomplished with Cedar covers a broad range, from basic computer graphics techniques, to various design tools with sophisticated graphical interfaces, to graphic-arts-quality typeset documents with embedded color illustrations. Our experience with Cedar confirms the benefits to a software researcher of shared module interfaces, compiler type-checking, automatic storage management, interpretive graphics programming languages, and device-independent imaging models. The 'object-oriented' programming style and the integration of graphics within Cedar, below the screen window manager and document formatter, have led to more effective software designs than those designed with traditional languages and programming environments. Cedar provides a software research environment where one quickly integrates the work of others and redesigns one's own work after experimenting with it in a functional prototype.

**CSL-84-7 July 1985**

**ON ADDING GARBAGE COLLECTION AND RUNTIME TYPES TO A STRONGLY-TYPED, STATICALLY-CHECKED, CONCURRENT LANGUAGE**
**by Paul Rovner**

Enough is known now about garbage collection, runtime types, strong-typing, static-checking and concurrency that it is

possible to explore what happens when they are combined in a real programming system.

Storage management is one of a few central issues through which one can get a good view of the design of an entire system. Tensions between ease of integration and the need for protection; between generality, simplicity, flexibility, extensibility and efficiency are all manifest when assumptions and attitudes about managing storage are studied. And deep understanding follows best from the analysis of systems that people use to get real work done.

This paper is not for those who seek arguments pro or con about the need for these features in programming systems; such issues are for other papers. This one assumes these features to be good and describes how they combine and interact in Cedar, a programming language and environment designed to help programmers build moderate-sized experimental systems for moderate numbers of people to test and use.

**CSL-85-1 February 1985**

**DISTRIBUTED NAME SERVERS: NAMING AND CACHING IN LARGE DISTRIBUTED COMPUTING ENVIRONMENTS**
**by Douglas Brian Terry**

Name services facilitate sharing in distributed environments by allowing objects to be named unambiguously and maintaining a set of application-defined attributes for each named object. Existing distributed name services, which manage names based on their syntactic structure, may lack the flexibility needed by large, diverse, and evolving computing communities. A new approach, structure-free name management, separates three activities: choosing names, selecting the storage sites for object attributes, and resolving an object's name to its attributes. Administrative entities apportion the responsibility for managing various names, while the name services information needed to locate an object's attributes can be independently reconfigured to improve performance or meet changing demands.

An analytical performance model for distributed name services provides assessments of the effect of various design and configuration choices on the cost of name service operations. Measurements of Xerox's Grapevine registration service are used as inputs to the model to demonstrate the benefits of replicating an object's attributes to coincide with sizable localities of interest. Additional performance benefits result from clients' acquiring local caches of name service data treated as hints. A cache management strategy that maintains a minimum level of cache accuracy is shown to be more effective than the usual technique of maximizing the hit ratio: cache managers can guarantee reduced overall response times, even though clients must occasionally recover from outdated cache data.

**CSL-85-2 June 1985**

**ARCHITECTURAL ELEMENTS FOR BITMAP GRAPHICS**
**by Cary D. Kornfeld**

This dissertation examines two closely related aspects of bitmap graphics--methods for manipulating bitmap images and techniques for displaying these images. First, it looks at the implementation of three primitive operations useful when manipulating images--Mirror, Rotate and Translate (BitBlt). When implemented on conventional bitmap systems these fundamental operations require extensive memory and processor activity. Alternative architectures for bitmap graphic systems are explored. Two new architectural elements that significantly improve system performance are defined. The first, a RasterOp unit, combines four bitmap images under arbitrary merging operations at video data rates. The second element, called an Image Prism, generates orthogonal image transformations on bitmap images without requiring processor interaction. As a result, these fundamental operations can be performed several hundred times faster than before. VLSI implementations of each are described and their performance is reviewed.

The second thrust of this research focuses on the design and development of a research display device. Three major architectural components of traditional CRT-based bitmap

display systems--frame buffer, display controller and display device--are integrated into a single, exceptionally thin, flat display device of which several working prototypes have been built. The entire display device is constructed using conventional microelectronic IC fabrication processing technology so that all required electronics are integrated onto a single silicon wafer which is then coated with a layer of electrophoretic display material. Research topics explored include new techniques for controlling display material at low voltage and a novel approach to defect tolerant wafer scale VLSI design. The behavior of the fabricated displays is reviewed and analyzed.

**CSL-85-3 May 1985**

**SETTING TABLES AND ILLUSTRATIONS WITH STYLE**
**by Richard J. Beach**

This thesis addresses the problem of formatting complex documents with electronic tools. In particular, the two problems of incorporating illustrations and laying out tables are treated in depth. The notion of style, a way of maintaining consistency in a document, runs throughout the thesis. It helps manage the complexities of formatting both illustrations and tables. The thesis reviews the history of document composition systems, including early computer typesetting systems, document compilers, and integrated document composition systems. The concept of graphical style is introduced to extend the more traditional notion of document style to illustrations. The observation that graphical style does not adequately deal with the layout problem for illustrations leads to the investigation of a more concentrated layout problem for the special case of table formatting. A grid system is used to describe the table layout arrangement and a constraint solver provides the general layout engine for formatting tables as well as the basis for future interactive table design tools. Further research based on the style and table formatting models can be extended to mathematical typesetting and full page layout. A glossary of typesetting terms and an index to the thesis are provided to help the reader deal with the typographic terminology used in the thesis.

**CSL-85-4 June 1985**

**WHITEBOARDS: A GRAPHICAL DATABASE TOOL**
**by James Donahue and Jennifer Widom***

The 'Whiteboards' system is intended to be an electronic equivalent of the whiteboards that we have in our offices. A Whiteboard database has similar qualities of storing disparate collections of data and saving their spatial location in a window to help with organization. A Whiteboard database can contain references to arbitrary entities: text files, notes, programs, tools, pictures, etc. Whiteboards runs as an application in the Cedar programming environment.

*Cornell University Computer Science Department

**CSL-85-7 November 1985**

**A CACHING FILE SYSTEM FOR A PROGRAMMER'S WORKSTATION**
**by Michael D. Schroeder, David K. Gifford, and Roger M. Needham**

This paper describes a workstation file system that supports a group of cooperating programmers by allowing them both to manage local naming environments and to share consistent versions of collections of software. The file system has access to the workstation's local disk and to remote file servers, and provides a hierarchical name space that includes the files on both. Local names can refer to local files or be attached to remote files. Remote files, which also may be referred to directly, are immutable and cached on the local disk. The file system is part of the Cedar experimental programming environment at Xerox PARC and has been in use since late 1983.

**CSL-85-8 November 1985**

**AN EFFECTIVE TEST STRATEGY**
**by Howard Sturgis**

In this paper I describe a debugging strategy that I have successfully used for several years. The principal idea is to excise test subjects from large programs and test them in individually crafted test beds, where the test beds are constructed according to four principles: (1) provide an "encapsulation" of the test subject that presents deterministic behavior to the test driver, (2) write a program to simulate the behavior of the encapsulated test subject, (3) use a random-number generator to construct the test sequences and compare the resulting behavior of the encapsulated subject with that of the simulation, and (4) provide the ability to exactly repeat a test sequence so as to isolate a detected bug through "binary chop."

While this strategy does not have the theoretical completeness provided by verification, I have found it to be easy to implement and considerably more effective than haphazard debugging. In all cases where I have used it, it has required much less time to construct the test beds than required for the original design and implementation of the program being tested. Further, I generally find that the resulting program is about as bug-free as it would have been had formal verification been available and used.

**CSL-85-9 November 1985**

**WALNUT: STORING ELECTRONIC MAIL IN A DATABASE**
**by James Donahue and Willie-Sue Orr**

Walnut is an electronic mail storage and retrieval system developed for the Cedar environment. The most novel aspect of Walnut is its use of a general-purpose relational database for storage of messages. This paper discusses the design and implementation of Walnut, with particular attention to the problems of using a database system for this type of application.

**CSL-86-1 June 1986**

## A STRUCTURAL VIEW OF THE CEDAR PROGRAMMING ENVIRONMENT

**by Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann**

This paper presents an overview of the Cedar programming environment, focusing on its overall structure - that is, the major components of Cedar and the way they are organized. Cedar supports the development of programs written in a single programming language, also called Cedar. Its primary purpose is to increase the productivity of programmers whose activities include experimental programming and the development of prototype software systems for a high-performance personal computer. The paper emphasizes the extent to which the Cedar language, with runtime support, has influenced the organization, flexibility, usefulness, and stability of the Cedar environment. It highlights the novel system features of Cedar, including *automatic storage management* of dynamically-allocated typed values, a *runtime type system* that provides runtime access to Cedar data type definitions and allows interpretive manipulation of typed values, and *a powerful device-independent imaging model* that supports the user interface facilities. Using these discussions to set the context, the page addresses the language and system features and the methodologies used to facilitate the integration of Cedar applications. A comparison of Cedar with other programming environments further identifies areas where Cedar excels and areas where work remains to be done.

**CSL-86-3 September 1986**

## VOICE ANNOTATION AND EDITING IN A WORKSTATION ENVIRONMENT

**by Stephen Ades* and Daniel C. Swinehart**

The Xerox Cedar experimental development environment, running on personal workstations, incorporates a structured document editor, which is used as the basis of many applications, including programming and document

preparation. A project at the Xerox Palo Alto Research Center has integrated the telephone into this environment, partly to afford control over a user's telephone from his workstation and partly to incorporate recording and playback of voice into the workstation capability.

This paper describes incorporation of voice annotations into normal documents within this environment. The user interface is designed to be lightweight and easy to use, since spontaneity in adding vocal annotations is essential. Voice within a document is shown as a distinctive shape superimposed around a character, so that the document's visual layout and its contents as observed by other programs (e.g., compilers) are unaffected. Users point at text selections and use menus to add and listen to voice.

Simple voice editing is available: users can select a voice annotation and open a window showing its sound profile. Sounds from the same or other voice windows can be cut and pasted together, and a lightweight "dictation facility" that uses a record/stop/backup model can be used to record and incorporate new sounds conveniently. Editing is done largely at the phrase level (never at the phoneme level), representing the granularity at which editing can be done fastest and with least effort. The voice itself can be annotated with text. This and several other features have been designed to add speed and meaning to the editing process. The dictation facility can also be used when placing annotations straight into documents.

This paper describes the user interface in detail and explains why we believe that this unusually lightweight interface is the desired abstraction for voice in a general purpose workstation. It also discusses the supporting system capabilities provided by a distributed environment on a local area network.

*University of Cambridge Computer Laboratory

**CSL-86-4 September 1986**

**A CLIENT INTERFACE TO AN ENTITY-RELATIONSHIP DATABASE SYSTEM**
**by James Donahue, Carl Hauser, and Jack Kent**

This paper presents the design of an interface, written in the Cedar language, for an entity-relationship data model database system (the database system is called Cypress). We discuss some of the design decisions that need to be made when building such an interface. This interface has been used to implement a number of database applications in the Cedar environment.

The material presented in this paper can be seen in two lights. First, it continues the development of "database programming languages." The novel aspect of the work in this regard is its use of the type structure and interface definition facility of the Cedar language to make database access an integral part of the Cedar environment without requiring any syntactic extensions to the underlying language. The second (and more interesting) aspect of the work reported here is that we give a completely operational description of an entity-relationship data model. The paper discusses several advantages of this approach over a purely relational system.

**CSL-87-7 August 1987**

**REIMPLEMENTING THE CEDAR FILE SYSTEM USING LOGGING AND GROUP COMMIT**
**by Robert Hagmann**

The workstation file system for the Cedar programming environment was modified to improve its robustness and performance. Previously, the file system used hardware-provided labels on disk blocks to increase robustness against hardware and software errors. The new system does not require hardware disk labels, yet is more robust than the old system. Recovery is rapid after a crash. The performance of operations on file system metadata, e.g., file creation or open, is greatly improved.

The new file system has two features that make it atypical. The system uses a log, as do most database systems, to recover metadata about the file system. To gain performance, it uses group commit, a concept derived from high performance database systems. The design of the system used a simple, yet detailed and accurate, analytical model to choose between several design alternatives in order to provide good disk performance.

**CSL-88-1 July 1988**

**VLSI DESIGN AIDS: CAPTURE, INTEGRATION, AND LAYOUT GENERATION**
**by Richard Barth, Louis Monier, Bertrand Serlet, and Pradeep Sindhu**

This report presents the VLSI tools recently developed in the Computer Science Laboratory at Xerox PARC. Three aspects are emphasized throughout the report: strong integration of tools, capture of designer's intents at a high level of abstraction, and an original framework for layout generation. The backbone of the system allowing tight tool integration is a data structure for representing circuits. Its features include powerful operations, strong conceptual integrity, rich expressive power, and high extensibility. The system enables the capture of designs as a mix of parameterized schematics and programs. Designs described in this manner are dense and legible, because they capture abstractions instead of merely net lists. Arbitrary properties may annotate designs. In particular, layout annotations provide all the information necessary to produce final masks. The layout subsystem, called PatchWork, is a unified framework for integrating both basic and powerful layout generators. A variety of major chips has been built using this system, and several in-depth examples are presented.

**CSL-88-2 July 1988**

## MAINTAINING THE ILLUSION OF A FUNCTIONAL LANGUAGE IN THE PRESENCE OF SIDE EFFECTS

**by Howard E. Sturgis**

This paper describes and proves correct a method for implementing first-order functions in the presence of side effects. A user supplies a recursion equation for each desired function. This equation contains a purely functional first-order expression involving both the desired functions and certain primitive functions. The desired functions are formally defined by the least fixed point of a call-by-value functional that is derived from the recursion equations. The user also provides an implementation of each primitive function (written in an imperative language), together with a declaration of the possible side effects caused by these implementations. A flow analysis algorithm is applied to the recursion equations in the context of the declared side effects. If the recursion equations pass the analysis, then they are translated into the imperative language of the primitive function implementations and executed together with them.

We prove that, if the recursion equations satisfy the flow analysis and the implementations of the primitive functions are both *faithful* and *safe*, then the resulting program computes the defined functions. The implementations of the primitive functions are faithful if they implement the intended primitive functions, and they are safe if they cause no undeclared side effects. We give a formal definition for faithful and safe. The proof is constructed by comparing two different computations: one in which the implementations of the primitive functions are free of side effects and one in which the implementations cause the declared side effects.

As a demonstration of the practicality of this method, we have implemented a compiler-compiler in which the compile function is defined by this method. The compiler-compiler is self-compiling.

**CSL-89-1 January 1989**

**EPIDEMIC ALGORITHMS FOR REPLICATED DATABASE MAINTENANCE**
**by Alan Demers, Mark Gealy, Dan Greene, Carl Hauser, Wes Irish, John Larson, Sue Manning, Scott Shenker, Howard Sturgis, Dan Swinehart, Doug Terry, and Don Woods**

When a database is replicated at many sites, maintaining mutual consistency among the sites in the face of updates is a significant problem. This paper describes several randomized algorithms for distributing updates and driving the replicas toward consistency. The algorithms are very simple and require few guarantees from the underlying communication system, yet they ensure that the effect of every update is eventually reflected in all replicas. The cost and performance of the algorithms are tuned by choosing appropriate distributions in the randomization step. The algorithms are closely analogous to epidemics, and the epidemiology literature aids in understanding their behavior. One of the algorithms has been implemented in the Clearinghouse servers of the Xerox Corporate Internet, solving long-standing problems of high traffic and database inconsistency.

**CSL-89-2 May 1989**

**ETHERPHONE: COLLECTED PAPERS 1987-1988**
**by Daniel C. Swinehart, Douglas B. Terry, and Polle T. Zellweger**

The Etherphone™ system integrates live and recorded voice into an office workstation environment. It supports a wide range of applications, including telephony, voice mail, voice annotation and editing, audio user interfaces, audio meeting services, and narrated hypermedia documents. This report brings together a group of previously-published papers that describe these applications and the distributed voice system architecture on which they are built. It includes an overview of the Etherphone system; a vision of the role that workstation-based telephones can play in an office environment; a discussion of the voice manager that provides

facilities for recording, editing, and playing stored voice; an outline of the systems support required to permit flexible, extensible multimedia applications; and a description of a hypermedia presentation system built upon the capabilities of the Etherphone system.

**An Overview of the Etherphone System and its Applications**, by Polle T. Zellweger, Douglas B. Terry, and Daniel C. Swinehart. This paper appeared in the Proceedings of the *2nd IEEE Conference on Computer Workstations* (Santa Clara, CA, March 1988), 160-168. An earlier version appeared as: D. Swinehart, D. Terry, and P. Zellweger. An experimental environment for voice system development. *IEEE Office Knowledge Engineering Newsletter*, **1**(1), February 1987, 39-48.

**Telephone Management in the Etherphone System**, by Daniel C. Swinehart. This paper appeared in the *Proceedings of the IEEE/IEICE Global Telecommunications Conference* (Tokyo, November 1987), 1176-1180.

**Managing Stored Voice in the Etherphone System**, by Douglas B. Terry and Daniel C. Swinehart. A version of this paper appeared in *ACM Transactions on Computer Systems*, **6**(1), February 1988, 3-27.

**System Support Requirements for Multi-media Workstations**, by Daniel C. Swinehart. This paper appeared in the *Proceedings of the SpeechTech '88 Conference* (New York; April 1988); Media Dimensions, Inc., New York, April 1988, 82-83.

**Active Paths through Multimedia Documents**, by Polle T. Zellweger. This paper appeared in *Document Manipulation and Typography*, J.C. van Vliet (ed.), Cambridge University Press, 1988. *Proceedings of the EP'88 Conference on Electronic Publishing, Document Manipulation and Typography*, (Nice, France; April 1988).

**CSL-89-3 January 1989**

**DATA COMPRESSION WITH FINITE WINDOWS**
**by Edward R. Fiala and Daniel H. Greene**

Several methods are presented for adaptive, invertible data compression in the style of Lempel's and Ziv's first textual substitution proposal. For the first two methods, the paper describes modifications of McCreight's suffix tree data structure that support cyclic maintenance of a window on the most recent source characters. A percolating update is used to keep node positions within the window, and the updating process is shown to have constant amortized cost. Other methods explore the tradeoffs between compression time, expansion time, data structure size, and amount of compression achieved. The paper includes a graph-theoretic analysis of the compression penalty incurred by our codeword selection policy in comparison with an optimal policy, and it includes empirical studies of the performance of various adaptive compressors from the literature.

**CSL-89-4 January 1989**

**UNIX NEEDS A TRUE INTEGRATED ENVIRONMENT: CASE CLOSED**
**by Mark Weiser, L. Peter Deutsch*, and Peter B. Kessler**

Long before there was CASE as we now know it, there were integrated programming environments for languages like Smalltalk and Lisp. Why are there no truly integrated environments for UNIX? And why is CASE not enough?

*ParcPlace Systems

**CSL-89-6 January 1989**

**EFFICIENT BINARY SPACE PARTITIONS FOR HIDDEN-SURFACE REMOVAL AND SOLID MODELING**
**by Michael S. Paterson* and F. Frances Yao**

We consider schemes for recursively dividing a set of geometric objects by hyperplanes until all objects are separated. Such a binary space partition, or BSP, is naturally considered as a binary tree where each internal node corresponds to a division. The goal is to choose the hyperplanes properly so that the size of the BSP, i.e., the number of resulting fragments of the objects, is minimized. For the two-dimensional case, we construct BSPs of size $O(n \log n)$ for n edges, while in three dimensions, we obtain BSPs of size $O(n^2)$ for $n$ planar facets and prove a matching lower bound of $\Omega(n^2)$. Two applications of efficient BSPs are given. The first is an $O(n^2)$-sized data structure for implementing a hidden-surface removal scheme of Fuchs, Kedem and Naylor [6]. The second application is in solid modeling: given a polyhedron described by its $n$ *faces*, we show how to generate an $O(n^2)$-sized CSG (*constructive-solid-geometry*) formula whose literals correspond to half-spaces supporting the faces of the polyhedron. The best previous results for both of these problems were $O(n^3)$.

*University of Warwick Department of Computer Science

**CSL-89-7 October 1989**

**BROWSING ELECTRONIC MAIL: EXPERIENCES INTERFACING A MAIL SYSTEM TO A DBMS**
**by Jack Kent, Douglas Terry, and Willie-Sue Orr**

A database management system provides the ideal support for electronic mail applications. The Walnut mail system built at the Xerox Palo Alto Research Center was recently redesigned to take better advantage of its underlying database facilities. The ability to pose ad-hoc queries with a "fill-in-the-form" browser allows people to browse their mail quickly and effectively, while database access paths guarantee fast retrieval of stored information. Carèful consideration of the systems'

usage was reflected in both the database schema representation and the user-interface for browsing mail.

**CSL-89-8 June 1989**

**EXPERIENCES CREATING A PORTABLE CEDAR**
**by Russ Atkinson, Alan Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser**

Cedar is the name for both a language and an environment in use in the Computer Science Laboratory at Xerox PARC since 1980. The Cedar language is a superset of Mesa, the major additions being garbage collection and runtime types. Neither the language nor the environment was originally intended to be portable, and for many years ran only on D-machines at PARC and a few other locations in Xerox. We recently re-implemented the language to make it portable across many different architectures. Our strategy was, first, to use machine-dependent C code as an intermediate language, second, to create a language-independent layer known as the Portable Common Runtime, and third, to write a relatively large amount of Cedar-specific runtime code in a subset of Cedar itself. By treating C as an intermediate code we are able to achieve reasonably fast compilation, very good eventual machine code, and all with relatively small programmer effort. Because Cedar is a much richer language than C, there were numerous issues to resolve in performing an efficient translation and in providing reasonable debugging. These strategies will be of use to many other porters of high-level languages who may wish to use C as an assembler language without giving up either ease of debugging or high performance. We present a brief description of the Cedar language, our portability strategy for the compiler and runtime, our manner of making connections to other languages and the Unix operating system, and some measures of the performance of our "Portable Cedar".

**CSL-89-9 August 1989**

**FLOATING-POINT AND COMPUTER SYSTEMS**
**by David Goldberg**

Floating-point is considered an esoteric subject by many people. This is rather surprising, because floating-point is ubiquitous in computer systems. Almost every language has a floating-point datatype, computers from PC's to supercomputers have floating-point accelerators, most compilers will be called on to compile floating-point algorithms from time to time, and almost every operating system has to perform some action when floating-point exceptions like overflow occur. This paper surveys aspects of floating-point that are likely to be useful to designers and users of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating-point standard, and concludes with numerous examples of how computer system builders can better support floating-point.

**CSL-89-10 November 1989**

**LARGE SCALE ANALYSIS OF NEURAL STRUCTURES**
**by Ralph C. Merkle**

Advances in computer analysis of images, the dropping cost of computer power and advances in light and electron microscopy and possibly in staining techniques will make it possible in the next few years to analyze neural structures of unprecedented size at the cellular level. A complete analysis of the cellular connectivity of a structure as large as the human brain is only a few decades away.

**CSL-90-1 February 1990**

**CONSTRAINED QUANTIFICATION IN POLYMORPHIC TYPE ANALYSIS**
**by Pavel Curtis**

Sound type systems have existed for several years for languages with polymorphism, the ability for procedures to

work on many kinds of data. Successful systems also exist for languages with a nontrivial notion of subtyping, such as is provided in some modestly object-oriented languages. The work in this dissertation was motivated by the problem of providing a sound and flexible type system for languages with both of these properties. We have developed a new kind of type expression, which we call constrained quantification, given a semantics for these new expressions, and developed algorithms for soundly inferring such types for programs in a particular sample programming language. This language has all of the essential features of a polymorphic functional language along with the kernel functionality of object-oriented languages.

In addition to solving the type system problem for this class of languages, constrained quantification outperforms the traditional polymorphic type analysis algorithms even in languages without subtyping.

We claim that constrained quantification is a flexible, powerful, practical and formally comprehensible approach to polymorphic type analysis, especially when applied to languages with nontrivial subtyping. The dissertation demonstrates this claim in more than enough detail for constrained quantification to find immediate use in a practical programming environment.

**CSL-90-2 April 1990**

**REBUILDING DATABASE CACHES DURING FAST CRASH RECOVERY**
**by Robert B. Hagmann**

Most database systems use a disk log for fast crash recovery. This paper proposes changes to the standard logging and recovery algorithms to make them faster, and, in addition, to recover various caches at little cost. One interesting cache is the buffer pool. For large configurations, the buffer pool (including old clean pages) is rebuilt from the log. During recovery, essentially all disk I/0 is the sequential reading of the log. With enough main memory, no random reads or writes to the disk database occur during recovery.

**CSL-90-3 April 1990**

**A MODULE SYSTEM FOR SCHEME**
**by Pavel Curtis and James Rauen***

This paper presents a module system designed for large-scale programming in Scheme. The module system separates specifications of objects from their implementations, permitting the separate development, compilation, and testing of modules. The module system also includes a robust macro facility.

We discuss our design goals, the design of the module system, implementation issues, and our future plans.

*MIT Laboratory for Computer Science

**CSL-90-4 June 1990**

**COMPARING STRUCTURALLY DIFFERENT VIEWS OF A VLSI DESIGN**
**by Mike Spreitzer**

One of the major problems of VLSI design is coping with the quantity and complexity of the design data. The leading solutions use 'divide-and-conquer' techniques. Two different ways of dividing are popular: division by a structural hierarchy, and division into various levels of abstraction (a *view* is a description at a particular level of abstraction). VLSI designs are so large and complex that both divisions are needed, which raises a question: should all the views of a design use the same hierarchy? This question is currently controversial. This dissertation, while not presuming to settle that question, argues in favor of allowing the views to have different hierarchies, and addresses a problem that is complicated by differences in hierarchy. That is the comparison problem, which has two parts: (1) verify consistency between alternate views, and (2) determine the correspondence between the design entities of those views. Previously existing techniques either work on flat views (that is, ones not divided into a hierarchical structure), or can only compare views that have

essentially identical hierarchies. Of course any hierarchical description can be flattened, but flattening is disadvantageous for a number of reasons. The most important reason is that flattening can exponentially increase the size of the description. Many comparison techniques require an amount of time that grows exponentially with the size of the circuit descriptions. Flat comparison techniques are thus impractical for VLSI designs.

This dissertation introduces a new comparison method, *Informed Comparison*, which neither requires the views to have essentially identical hierarchies nor flattens the views. Informed Comparison requires the designers to maintain a *key*, which is a description of the intended relation between the hierarchies of the views. Informed Comparison first *reconciles* copies of the views by applying hierarchy transformations, under the guidance of the key, until the copies have essentially identical hierarchies. Informed Comparison then finishes with a *base comparison*, which can use any existing (or new) hierarchical technique that assumes essentially identical hierarchies. Informed Comparison thus has many of the good features, including good asymptotic performance, of other hierarchical methods.

Several characteristics of Informed Comparison depend on the repertoire of transformations available to the reconciliation step and on the base comparison technique. This dissertation illustrates those dependencies with two examples of Informed Comparison.

**CSL-90-5 June 1990**

**AN ARCHITECTURE FOR HIGH-PERFORMANCE SINGLE-CHIP VLSI TESTERS**
**by James A. Gasbarro**

Testing is an important factor in the production of useable custom integrated circuits. Verification of the functional and AC parametric characteristics of a device are usually performed on large and expensive test systems. This thesis presents a new approach to tester architecture that seeks to greatly reduce both the size and cost of these systems. The principal

idea is to base the tester design on the same high-density technology as that of the devices being tested. Through the use of novel test vector compression techniques and closed-loop timing calibration methods, high performance and high density can be achieved in a CMOS technology. The proof is the implementation of a single-chip multi-channel tester which has the size and cost attributes of the very low-end testers, yet implements many of the features found on only the most expensive machines.

The high level of integration achieved results in a number of other advantages as well. The close proximity of the tester to the test device eliminates most of the signal transmission and loading issues encountered in larger systems. The extremely compact size enables in-circuit probing and performance analysis of the test device without custom fixturing. Finally, and perhaps most importantly, by implementing the tester in the same technology as that of the device to be tested, future upgrades of the tester capability can evolve along with the capabilities of the subject device.

**CSL-90-6 June 1990**

**ACTIVE TIOGA DOCUMENTS**
**AN EXPLORATION OF TWO PARADIGMS**
**by Douglas B. Terry and Donald G. Baker***

The advent of electronic media has changed the way we think about documents. Documents with illustrations, spread sheets, and mathematical formulae have become commonplace, but documents with active components have been rare. This paper focuses on our extensions to the Tioga editor to support two very different styles of active documents. One paradigm involves dynamically computing, or at least transforming, the contents of a document as it is displayed. A second paradigm uses notifications of edits to a document to trigger activities. Document activities can include database queries, which are evaluated and placed in the document upon opening the document, or constraints between portions of a document, which are maintained as the user edits the document. The resulting active documents can

be viewed, edited, filed, and mailed in the same way as regular documents, while retaining their activities.

*Rice University

**CSL-90-7 August 1990**

**HIGHLY PARALLEL SPARSE CHOLESKY FACTORIZATION**
**by John R. Gilbert and Robert Schreiber***

We develop and compare several fine-grained parallel algorithms to compute the Cholesky factorization of a sparse matrix. Our experimental implementations are on the Connection Machine, a distributed-memory SIMD machine whose programming model conceptually supplies one processor per data element. In contrast to special-purpose algorithms in which the matrix structure conforms to the connection structure of the machine, our focus is on matrices with arbitrary sparsity structure. Our most promising algorithm is one whose inner loop performs several dense factorizations simultaneously on a two dimensional grid of processors. Virtually any massively parallel dense factorization algorithm can be used as the key subroutine. The sparse code attains execution rates comparable to those of the dense subroutine. Although at present architectural limitations prevent the dense factorization from realizing its potential efficiency, we conclude that a regular data parallel architecture can be used efficiently to solve arbitrarily structured sparse problems.

We also present a performance model and use it to analyze our algorithms. We find that asymptotic analysis combined with experimental measurement of parameters is accurate enough to be useful in choosing among alternative algorithms for a complicated problem.

*Research Institute for Advanced Computer Science, NASA Ames Research Center

**CSL-90-8 August 1990**

**SEPARATORS IN GRAPHS WITH NEGATIVE OR MULTIPLE VERTEX WEIGHTS**
**by Hristo N. Djidjev* and John R. Gilbert**

A separator is a small set of vertices whose removal divides a graph approximately in half. Separator theorems are known for graphs that forbid a given minor (including planar graphs, for example); finite element graphs; chordal graphs; and some others.

Here we show that a separator theorem implies various weighted separator theorems. If real-valued (possibly negative) vertex weights are given, the graph can be divided exactly in half by weight. If two unrelated sets of positive weights are given, the graph can be divided by both weights simultaneously.

*Center of Informatics and Computer Technology, Bulgarian Academy of Sciences

**CSL-90-9 November 1990**

**OPTIMAL EXPRESSION EVALUATION FOR DATA PARALLEL ARCHITECTURES**
**by John R. Gilbert and Robert Schreiber***

A data parallel machine represents an array or other composite data structure by allocating one processor (at least conceptually) per data item. A pointwise operation can be performed between two such arrays in unit time, provided their corresponding elements are allocated in the same processors.

If the arrays are not aligned in this fashion, the cost of moving one or both of them is part of the cost of the operation. The choice of where to perform the operation then affects this cost. If an expression with several operands is to be evaluated, there may be many choices of where to perform the intermediate operations. We give an efficient algorithm to find the minimum-cost way to evaluate an expression, for

several different data parallel architectures. Our algorithm applies to any architecture in which the metric describing the cost of moving an array has a property we call "robustness." This encompasses most of the common data parallel communication architectures, including meshes of arbitrary dimension and hypercubes. We remark on several variations of the problem, some of which we solve and some of which remain open.

*Research Institute for Advanced Computer Science, NASA Ames Research Center

**CSL-90-10 January 1991**

**APPROXIMATING TREEWIDTH, PATHWIDTH, AND MINIMUM ELIMINATION TREE HEIGHT**
**by Hans L. Bodlaender*, John R. Gilbert, Hjálmtyr Hafsteinsson†, Ton Kloks***

We show how the value of various parameters of graphs connected to sparse matrix factorization and other applications can be approximated using an algorithm of Leighton et al. that finds vertex separators of graphs. The approximate values of the parameters, which include minimum front size, treewidth, pathwidth, and minimum elimination tree height, are no more than $O(\log n)$ (minimum front size and treewidth) and $O(\log^2 n)$ (pathwidth and minimum elimination tree height) times the optimal values. In addition we examine the existence of bounded approximation algorithms for the parameters, and show that unless $P = NP$ there are no absolute approximation algorithms for them.

*University of Utrecht
†University of Iceland

**CSL-90-11**

**ELIMINATION STRUCTURES FOR UNSYMMETRIC SPARSE *LU* FACTORS**
**by John R. Gilbert and Joseph W.H. Liu***

The elimination tree is central to the study of Cholesky factorization of sparse symmetric positive definite matrices. In this paper, we generalize the elimination tree to a structure appropriate for the sparse *LU* factorization of unsymmetric matrices. We define a pair of directed acyclic graphs called *elimination dags,* and use them to characterize the zero-nonzero structures of the lower and upper triangular factors. We apply these elimination structures in a new algorithm to compute fill in sparse *LU* factorization. Our experimental results indicate that the new algorithm is usually faster than existing methods.

*Department of Computer Science, York University

**CSL-90-12 September 1990**

**7 STEPS TO A BETTER MAIL SYSTEM**
**by Douglas Terry**

Electronic mail systems were developed as a means for sending interpersonal messages between users. Distribution lists in current mail systems are being increasingly used for disseminating information to large groups of readers. Mail systems are not ideally suited for this role. This paper proposes seven steps that can be taken to improve our mail systems. The intent is to allow more effective communication among groups of people. The steps collectively lead to a non-traditional mail system architecture, one utilizing a common storage system and recipient-controlled message delivery.

**CSL-90-13 December 1990**

**PHASE-SLIP TECHNIQUE FOR DIRECT SEQUENCE SPREAD SPECTRUM COMMUNICATION**
**by Edward A. Richley and Richard M. Barth***

The advent of portable workstations has created a need for wireless local area networks. In addressing these needs, many techniques have appeared recently to take advantage of the new FCC rules regarding the use of spread spectrum on the 902-928 Mhz and other, higher, bands. Most of these techniques offer degraded performance or excessive power requirements in order to comply with the rules. A technique, referred to as "Phase-Slip-Locking" has been devised to circumvent these shortcomings. A direct-sequence spread spectrum system utilizing the entire available process gain has been built in prototype form. Most importantly, the power consumption requirement of the receiver is only slightly more than that for a conventional narrow band system. The system is operated at a center frequency of 915 Mhz, with a transmitted power level of 100mW. Although much refinement is yet to be performed, these initial tests indicate that the Phase-Slip technique is very useful.

*Rambus, Inc.

# AUTHOR INDEX

## G



## H



## I



## J



## K

**P**



**R**



**S**



**T**

**W**



**Y**



**Z**