Final Report on Information Processing Research

July 1987 to July 1990

Carnegie Mellon University School of Computer Science

This research is sponsored by the Defense Advanced Research Projects Agency, DoD, through DARPA order 4976, and monitored by the Air Force Avionics Laboratory under contract F33615-87-C-1499. Views and conclusions in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the United States Government.

INTRODUCTION

This report documents a broad program of basic and applied information processing research conducted by Carnegie Mellon's School of Computer Science. The Information Processing Technology Office of the Defense Advanced Research Projects Agency (DARPA) supported this work during the period 15 July 1987 through 14 July 1990, and extended the contract to 31 December 1990.

Chapters 1 through 7 present in detail our seven major research areas: Artificial Intelligence, Image Understanding, Reliable Distributed Systems, Programming Environments, Reasoning About Programs, Uniform Workstation Interfaces, and Very Large Scale Integration. Sections in each chapter present the area's general research context, the specific problems we addressed, our contributions and their significance, and a bibliography for each chapter.

1. Work statement

We organize the research reported here under seven major headings. These interrelated projects and their major objectives are:

Artificial Intelligence

Perform basic research in artificial intelligence, with emphasis on representation of knowledge, learning, and parallelism for AI applications. Specific subtasks include:

- Continue developing Soar, a domain-independent architecture for intelligent behavior. Research directions include extending chunking as a learning mechanism, developing planning mechanisms, and commencing at least one major task domain.
- Develop interactive learning models in which learning takes place automatically through interactions with a complex environment and also by taking direct instruction from humans (Prodigy).
- Continue to explore the interaction of knowledge and search in the context of the chess machine and in at least one other application domain.
- Explore the concurrency available in classes of expert systems tasks, relate this to the structure of parallel production systems, and incorporate parallel-decomposition techniques into the knowledge-acquisition tools that are being developed elsewhere at CMU (in conjunction with PSM).

Image Understanding

Perform basic research in image understanding, emphasizing knowledge representation and algorithm acquisition for vision systems. Specific tasks include:

- Continue developing new computational methods for inferring surface and shape information by exploiting image color, texture, and motion.
- Building on a production-system foundation, develop a knowledge-based vision-system framework that integrates high- and low-level vision and allows image-analysis systems to employ symbolic, geometric, and image-feature reasoning efficiently.

- Develop techniques for automatically acquiring 3-D object recognition algorithms from CAD-type shape descriptions and sample images. Demonstrate the techniques in a practical application such as a hand-eye, bin-picking system.

Reliable Distributed Systems

Develop a system that supports reliable, distributed applications on networks of uniprocessors and shared memory multiprocessors. Specific subtasks include:

- Construct a transaction-based facility that provides communication, recovery, and synchronization for distributed applications (Camelot).
- Design and implement language facilities for Ada and Common Lisp to provide application programmers access to Camelot and Mach facilities (Avalon).
- Develop distributed algorithms to support applications requiring high reliability and availability.
- Demonstrate entire Reliable Distributed System (RDS) on selected applications.

Programming Environments

Design and implement mechanisms for software development environments that can evolve incrementally while supporting large, cooperative, development projects. Specific subtasks include:

- Develop an environment kernel that provides uniform tool interfaces and allows new tools to be incorporated into an integrated environment.
- Develop techniques to support project coordination, to include automatic propagation of information and enforcement of policies.
- Develop concepts and tools to support large projects in a heterogeneous development environment with hierarchies of policies.

Reasoning about Programs

Perform basic research on the semantics of programming concepts, and develop semantically-based tools for reasoning about programs. Specific subtasks include:

- Investigate the semantic foundations of programming languages, with the aim of designing semantically-based proof systems for reasoning about program properties.
- Design and implement interactive program proof systems to be integrated into an advanced programming environment (Ergo).
- Develop manageable and powerful proof methods for dealing with concurrent programs.

Uniform Workstation Interfaces

Develop a uniform workstation interface system for a heterogeneous distributed computing environment. This interface system is to be integrated with Mach and will become an integral part of the Mach-based environment. Specific subtasks include:

- Develop an interface manager through which all interactions take place and that supports multiple interaction styles simultaneously.
- Design and implement a system for modeling knowledge about the system and the user, and use as a basis for an adaptive help system.
- Develop a language-independent application interface that can accommodate the needs of all the different types of applications, from batch-oriented to highly interactive ones.
- Demonstrate the above features in a highly efficient integrated system for a large number of application programs, and evaluate this system in a large user community at CMU.

VLSI

Develop methodologies and tools for rapidly building and validating VLSI systems. Specific subtasks include:

- Develop design environments and associated methodologies for building special purpose architectures, and demonstrate by building prototype systems.
- Develop high-performance switch-level and symbolic simulators, including parallel implementations of these simulators.
- Develop verification methodologies and tools capable of handling both synchronous and asynchronous circuits and hierarchically constructed circuits.

1. RESEARCH IN ARTIFICIAL INTELLIGENCE

Our research in artificial intelligence aims at improving the performance of AI systems when compared to the highest human level of performance. We also aim to reduce the time needed to solve tasks. The fundamental understanding of AI resulting from this research will provide the necessary foundation for applying AI to complex DoD tasks. This project includes a number of efforts exploring new directions in AI research:

- Developing expert systems that automatically acquire knowledge from large databases, concentrating on the domain of aerial image interpretation
- Developing systems that solve very difficult problems by the heuristic search of huge spaces (tens of millions of nodes) using special hardware architectures and exploring ways of guiding this fast, hardware-assisted search with high-level knowledge
- Exploring the use of massively parallel, connectionist ("neural network") architectures for learning, knowledge representation, recognition, and symbolic processing
- Developing computer systems that can solve problems and learn in a wide range of complex tasks by integrating learning and problem solving into single systems.

1.1. Task-level parallelism for production systems

Large production systems (rule-based systems) continue to suffer from extremely slow execution which limits their utility in practical applications as well as research settings. Most efforts at speeding up these systems have focused on match or knowledge-search parallelism in production systems. Though good speed-ups have been achieved in this process, the total speed-up available from this source is not sufficient to alleviate the problem of slow execution.

We focus on task-level parallelism, which is obtained by a high-level decomposition of the production system. For the familiar OPS5 production system computational model, task-level parallelism allows multiple rules in the production system to be fired in parallel. Speed-ups obtained from task-level parallelism are distinct from those obtained from match parallelism, and the two can be combined to provide even faster performance.

Our vehicle for the investigation of task-level parallelism is SPAM (System for Photointerpretation of Airports using MAPS), a high-level vision system implemented in a production system architecture. SPAM tests the hypothesis that the interpretation of aerial imagery requires substantial knowledge about the scene under consideration. SPAM has been applied in two task areas: airport and suburban house scene analysis. SPAM is a mature research system having over 600 productions, with a typical scene analysis task having between 50,000 to 400,000 production firings and an execution time of the order of 20-120 CPU hours (when measured with the Lisp OPS5 version VPS2 running on a VAX 11/785).

In investigations of parallelism, it is important to speed up an optimized sequential implementation, otherwise the benefits of parallelism are lost. Hence, SPAM was first reimplemented in ParaOPS5, a C-based optimized implementation of OPS5, that extracts match parallelism. The reimplementation required some minor changes to the existing OPS5 productions. We will refer to this reimplementation as SPAM/PSM. This provided a factor of about 15-20 in speedup—some of this from better internal indexing mechanisms in ParaOPS5, and some from the change from Lisp to C. However, match parallelism of ParaOPS5 provided only a factor of 1.5-2 in additional speedups. This is because, unlike most production systems, SPAM/PSM spends only 50% of its execution time in the match phase, thus putting an upper bound of two on the speedups obtained from match.

1.1.1. Task-Level parallelism in SPAM/PSM

The difficulties of extracting match parallelism encouraged our investigation of task-level parallelism in SPAM/PSM. We have currently limited our exploration of task-level parallelism in SPAM/PSM to two particular phases called LCC (local consistency check), and RTF (region to fragment). The choice of LCC was motivated by the observation that it consumes more than 90% of SPAM/PSM's execution time. The RTF phase was selected for parallelization since it fits the framework of a traditional OPS5-system more closely than the other phases of SPAM—it thus contrasts with the computation in LCC, providing generality to the results presented. To describe our approach to task-level parallelism and how it differs from some other approaches reported in the literature, we first present a characterization of task-level parallelism in production systems along three dimensions:

- *Implicit/Explicit:* In the implicit approach to extraction of parallelism, it is the compiler's job to extract parallelism; in the explicit approach, the user specifies the information for exploiting task-level parallelism.
- *Synchronous/Asynchronous:* Production systems can either fire productions asynchronously, or force synchronization of productions. Synchronous systems are less capable of handling variances in processing times of subtasks than asynchronous systems.
- *Rule distribution/working memory distribution:* In rule-distribution, the productions in the system are distributed among processors, where each production set maintains its own conflict set. In working memory distribution, all productions are allocated to each processor, but the working memory is distributed.

The computation in the LCC and RTF phases of SPAM/PSM can be decomposed into independent subtasks at different granularities. This suggested the appropriateness of the explicit approach to task-level parallelism. An explicit approach also raises the question of an appropriate granularity of decomposition. This choice is based on the communication overheads, variance in subtask processing times and the ease of decomposition. For the LCC phase, we explored two different decompositions. Table 1-1 presents results from these decompositions with the SPAM/PSM system running in a uniprocessor mode. The first, called level 3, has about 50-200 independent subtasks available, each executing about 5 seconds. The second, called level 2, has about 400-1000 independent subtasks available, each executing about 1.5 seconds.

Table 1-1: Measurements for baseline system on the datasets (Represents the optimized, ParaOPS5-based, uniprocessor version).

Dataset	Total time (sec)	Number of tasks	Average time per task (sec)	Prods fired	RHS actions	Changes to Working Memory
SF Level 3	1433	283	5.07	33475	42383	39116
SF Level 2	1423	941	1.51	32251	41159	38550
DC Level 3	988	151	6.55	20059	31205	26714
DC Level 2	956	490	1.95	19418	30564	26412
MOFF Level 3	991	209	4.74	22203	23637	23368
MOFF Level 2	973	700	1.39	21294	22728	22950

With the explicit approach selected for task-level parallelism, no synchronization was required, leading to the use of an asynchronous approach. Finally, we chose working memory distribution, as that facilitates the explicit decomposition. Thus, for SPAM/PSM (both the LCC phase and the RTF phase) we have an explicit/asynchronous/working-memory-distribution approach to parallelization. This contrasts with the other approaches to task-level parallelism reported in the literature, which have been implicit/synchronous approaches.

1.1.2. Results of the SPAM/PSM Implementation

Our experiments on task-level parallelism in SPAM/PSM were performed on the Encore Multimax, a 16-processor shared-memory multiprocessor based on the NS32332 processors. The experiments were performed with three different large data sets. These data sets represent three different airports: SF (San Francisco International), DC (Washington National), MOFF (NASA Ames Moffett Field). The speedup curves show near linear speedup for both the level 3 and level 2 decompositions. The maximum speedup achieved is 12-fold in level 3 and 12.6-fold at level 2 with 14 processors. These results also indicate that the potential for additional speedups in SPAM/PSM from task-level parallelism is quite high. In fact, an expectation of 50- to 100-fold does not seem unreasonable, given the number of independent subtasks available and the fairly coarse grain size of execution. In fact, at present, the scheduling overhead of task-level parallelism is less than 0.1% of the processing time. Similar speedups were observed for the RTF phase.

With the substantial amounts of parallelism available in SPAM/PSM, the 16-processor Encore Multimax appeared inadequate for exploring this. We explored two different approaches to alleviate this problem:

- The use of network shared memory: This technology will potentially allow us to share the processors on two different Encore Multimax machines, giving us a total of 32 processors to experiment with. While the technology is not very mature at this point, it has already shown promising results by extending the speedup from 12-fold to 15-fold with the total of 22 processors over the two Encores. We are continuing investigation of the use of network shared memory.

- Message-passing computers: An alternative to the shared-bus shared-memory technology used in the Encore Multimax is the use of more easily scalable distributed memory machines. We have explored the use of such machines for parallel production systems. These explorations have also shown promising results, indicating that the communication overheads are limited. In fact, for our benchmark systems, simulations indicate speedups similar to the shared-bus machines like the Encore Multimax. Given man-power limitations, and the fact that network shared memory technology was more readily available, we have focused on network shared memory for our actual implementation. However, the use of message-passing computer technology remains an interesting issue for future explorations.

Our experiments also verified that match parallelism does represent an independent axis of parallelism and thus could be multiplied with the speed-up obtained from task-level parallelism. In the three datasets tested, this axis provided an additional factor of 1.5- to 2-fold parallelism. Although for SPAM/PSM match parallelism does not seem to contribute as much, for more match-intensive applications, match parallelism will make a substantial contribution to the speed-ups.

1.2. Search-intensive AI systems

In the quest for higher performance, "more of the same" frequently offers diminishing amounts of "better." For AI systems, this means that it would be more productive to blend more search into a knowledge-intensive system, or more knowledge into a search-intensive system. We are studying the latter. Our research offers an opportunity to study the effects of *extremely* fast—and relatively clever—searches in very large problem spaces. The opportunity here is significant because we have no experience with intelligent systems, human or mechanical, that solve problems in this manner.

Humans are known to solve certain problems in what could be called the knowledge-intensive mode, with a small amount of search being invoked when necessary. This provides the flexibility that is needed to avoid encoding *all* knowledge. Many current AI systems mimic this. Indeed, the explosion of expert systems can be seen as an attempt to exploit the knowledge-intensive style with as little problem search as possible. However, it is clear that most problems can be solved by trading off search for knowledge and vice versa. If one considers a two-dimensional space of search and knowledge, hyperbola shaped isobars exist that define equal performance levels made up of varying amounts of knowledge and search.

It is typically difficult to provide the power to do really large searches. However, when the opportunity presents itself to study the effect of very large (and fast) searches on problem solving, such opportunities should not be ignored. Human problem solving provides no experience with this manner of accomplishing tasks. Such a style does not mean the absence of immediate knowledge, rather it means using relatively small amounts of knowledge to modulate and guide a large search.

Chess provides a particularly good laboratory in which to study search-intensive techniques. Chess programs and chess machines that search a large space of possibilities in a very simple way have established dominance over chess programs that try to bring a lot of knowledge to bear in guiding the search and evaluating each situation.

We have been using Hitech, our chess machine/program, to investigate search-intensive architectures. Hitech combines fast search hardware with extensive knowledge in the domain of chess, and incorporates techniques for adding new knowledge.

Our goals in this research area were to:

- Continue work on the interaction of knowledge and search in the context of the chess machine.
- Characterize the effect of deeper searches on the type of knowledge required to make good decisions.
- Extend the concept of chunking and characterize more clearly the kinds of problems where this idea is applicable.
- Identify practical real-world domains where the search-intensive approach used in the chess machine can be of value, and demonstrate this for at least one such domain.

1.2.1. Refinement

As of August 1987, the new knowledge in Hitech appeared to blend very well with the second-generation pattern recognizers that were installed in the last half of 1986. The installing of this knowledge took much longer than expected because the overall problem of providing knowledge that is both pertinent and correct is quite difficult. By the end of 1987, we had entered a mode of patching small areas of incompetence, with Hitech's knowledge in solid shape.

As we refined and augmented the pattern knowledge used in the course of a search, Hitech's performance continued to improve. In addition, we developed new search techniques that retain the efficiency of alpha-beta search while noting features that suggest exploring some lines of play more deeply than others. This makes it possible to explore certain critical lines to twice the normal depth [Anantharaman et al. 88]. We designed a new pawn structure algorithm and upgraded the king-safety pattern recognizers, which contributed significantly to Hitech's success. We supplemented the design with new pattern recognizing software to fill out certain knowledge gaps.

We have designed, built, and installed some new hardware to produce more sophisticated evaluation of pawn structures. This design was based on knowledge acquired by having a similar facility in programmable pattern recognizers. When we knew what we wanted, this was converted to permanent hardware which now frees up the programmable units to be used for other experiments.

Hitech's playing has improved continually during the contract period, proving that adding more knowledge continues to improve performance, even when search speed is a major factor. Examples of Hitech's knowledge include the ability to recognize progressively more complex board patterns, and to assign appropriate values that reflect a chess master's assessment of the patterns.

1.2.2. Results: Competition

Since August 1987, Hitech's performance has been outstanding. It won the 1987 Pennsylvania State Championship Tournament with 15 masters competing. In 1988, Hitech again won the Pennsylvania State Chess Championship tournament. In a field of 46 players, it scored 4.5 - 0.5 for a clear 1st place. With this victory, Hitech crossed the magical 2400 boundary and became a Senior Master in the US Chess Federation.

In a specially arranged match in September 1988 in New York City, Hitech crushed International Grandmaster and former US Champion Arnold Denker by the score of 3.5-0.5. This was the first time an International Grandmaster had been beaten by a computer. In another tournament, Hitech drew with a player who is ranked among the top 12 in the US, after having missed opportunities to beat him. Hitech is now among the top 150 players in the US.

In November, Hitech tied for 6th place in the National Open with a number of chess notables. This attracted wide attention, and was covered in Time Magazine.

In April 1989, Hitech drew with International Grandmaster J. Piket (rated FIDE 2500) in a match of Computers vs. the Netherlands Championship competitors. In May, Hitech finished second in the World Computer Chess Championships, behind Deep Thought, also from CMU. In the June Asian Action Team tournament, Hitech scored 7-3 on first board on a computer team, and placed 4th among the 11 competing first board players. In a June rematch against Manuel Apicella of France (FIDE 2400), Hitech scored 1.5 - 0.5. Hitech drew with Apicella (1-1) last year when his rating was FIDE 2365. This is a direct comparison since the human had improved in the meantime, and yet Hitech's performance was better.

In August 1989, Hitech won the Pennsylvania State Championship for the third time in a row with a perfect 5-0 score. In the 1989 North American Computer Championship, Hitech achieved first place on tie-breaking points, ahead of Deep Thought. Hitech has now reached the highest USCF rating in its career: 2413.

In May 1990 Hitech scored a significant victory in the AEGON International Tournament in the Netherlands. It scored 5-1, defeating a former World Champion Challenger and drawing with another Grandmaster. Its score was far above the other computers participating, the best of whom scored only 3-3. This event received tremendous coverage in the European press. In November Hitech beat Deep Thought, another Carnegie Mellon chess machine, in the ACM National championship.

1.2.3. Overview of Achievements

Massive Search has proven itself not only in chess, but also in Speech Recognition, and other application. With Hitech hardware we have succeeded in speeding up the solving process by about 3 orders of magnitude over what was available without a special purpose machine at the time Hitech was first built (1983-84). With the special purpose architecture many problems that were formerly out of reach could now be addressed. The addition of pattern recognizers that could run without slowing the machine down (since they are also hardware running in parallel) has made it possible to create very complex subgoals that the machine tries to achieve, helping Hitech to behave as if it is planning its moves.

The complexity of such subgoals is important; when one is competing at high levels of skill, simplistic goals will not produce useful behavior on the part of the machine. A simple-goal driven machine must rely solely on what it discovers during the search and what it can understand in terms of these simplistic goals. With complex patterns, progress becomes obvious much earlier. For example, if one can identify a weak pawn that may be lost, it will be not too big a surprise when the search finally finds a way of winning the pawn. Without the information on weak pawns, the search will not try to weaken a pawn (or avoid having one of its own weakened) and will be hugely surprised when such a pawn is won.

1.2.4. Application to other Domains

Similar machines can be built for any domain that is worth exploring in great detail. Candidate domains are usually of the structural variety, such as architectural design, chemical structure problems, design layout of computer chips. What is required is that the rules of the domain be known quite exactly, and that the number of possibilities to explore is beyond what can be achieved on a standard computer or costs too much on a supercomputer.

1.3. Using massively parallel architectures

The term "connectionism" refers to attempts to perform tasks such as recognition, learning, and knowledge representation using a large network of very simple neuron-like elements, all operating in parallel. The study of AI on serial architectures was already a well-developed field when we began our research; however, we were still in the early stages of understanding how to use connectionist networks. Our hope was that this approach would lead to recognition systems that can be trained rather than programmed, built on an inherently fault-tolerant hardware base. Carnegie Mellon has played a large part in the renaissance of this field, which has grown into a major and important area of research.

The DARPA basic research contract funded only a small part of our overall connectionist effort, specifically the development of new learning algorithms and simulation tools. Closely related work on autonomous navigation and on the use of the Warp machine as a simulation engine was not directly funded by DARPA, but made heavy use of facilities provided by DARPA.

1.3.1. Tools for connectionist research

In the first six months of this research, a great deal of time was spent on tool-building. We developed a variety of network simulation tools in both **C** and Common Lisp. In addition, we built a back-propagation simulator for the 10-processor, 100 MFLOPS Warp machine. We needed this speed (20 million connections per second) for big jobs.

A network display tool

We produced an animated display that follows the evolution of a small network (a two-input XOR). The display shows each network layer’s composite behavior, rather than just the states of individual units and weights. This tool enabled us to observe the layers’ composite behavior; often the interesting and valuable phenomena are not properties of individual components, but properties of component groups that may correlate behaviorally but be scattered across the network. This, initially a minor effort, provided insights into the "herd effect" that led to the development of the Cascade-Correlation learning algorithm.

A connectionist network simulator

We developed a connectionist network simulator on the 10-processor Warp machine that processes 20 million connections per second [Pomerleau et al. 88]. This is about 8 times the speed claimed for a 16K CM-1 connection machine, which is much more expensive, and over 300 times the speed obtainable on a machine of the VAX-780 class. The Warp-based connectionist simulator makes it possible to apply connectionist networks to much larger problems. We distributed the software, called the Warp Neural Network Toolkit, to other Warp sites. This work paved the way for later neural-net simulators on the experimental GF11 machine at IBM Research and on the iWarp processor, which is being produced commercially by Intel.

An international mailing list for connectionist researchers

We maintain an international electronic mailing list for connectionist researchers that now has thousands of readers (direct and indirect) on six continents. This is a very important communication medium for neural net researchers—probably as important as any journal, and certainly a better source of timely information. This is not a large effort, but one that has a real impact in the neural-net research community.

The online benchmark collection

We have established an online collection of benchmark problems and data sets for measuring neural net learning algorithms. For each problem, we maintain a summary of the best results so far reported in the literature. Such a collection is necessary to our own learning research, and it allows other researchers to compare their results in a responsible and scientific manner.

1.3.2. New Learning Algorithms

The most important problem limiting the widespread application of neural-nets to real-world problems is the slow learning speed and unreliability of existing learning algorithms such as back-propagation. These algorithms scale up poorly as we increase the size and complexity of the learning task. We have worked for several years to understand why back-propagation is so slow and what can be done about this.

The first round of exploration led to the development of the Quickprop algorithm. In place of the simple gradient descent that backprop uses to reduce the error during training of the network, Quickprop makes use of second-derivative information to control the size of the weight adjustments. On simple problems, this improves learning time by a factor of 10 to 100, with even more speedup on larger problems [Fahlman 88].

However, even the Quickprop algorithm is too slow for many practical applications. Further study revealed that the principal cause of this problem was in the uncontrolled dynamic behavior of the "hidden" or interior units in the network. These units interact only weakly with one another, and it takes a long time for them to settle into appropriate roles, with each unit doing a different job. We developed the Cascade-Correlation algorithm [Fahlman and Lebiere 90] to correct this problem. In Cascade-Correlation, we begin with a net that has only inputs, output units, and the connections between them. Then we add hidden units one by one. Each new unit moves quickly and directly to cancel as much of the remaining error as possible. Once the new unit has found a role to play, its input weights are frozen. As more hidden units are added, each one deals only with the remaining portion of the error.

The cascade-correlation approach has several advantages over the older backprop and quickprop models. Users no longer have to guess what network size and shape to use for a problem, because the new method builds a near-minimal network automatically. Since already-created, hidden-unit feature detectors are never altered, we can teach the network new behaviors without destroying the structure of previously learned behaviors. In learning experiments to date, the training time appears proportional to the number of hidden units ultimately needed for the task. On the very difficult two-spirals benchmark, cascade-correlation reliably solves the problem 25 times faster than Quickprop and 50 to 100 times faster than standard backprop.

1.3.3. Autonomous Navigation

Using the connectionist network simulator on Warp, we developed a connectionist road following system (ALVINN) that has successfully driven the Navlab vehicle on a path through a wooded area. The road following network's performance is comparable to that obtained by standard methods. Navlab now travels at greater than its previous 0.5 m/s speed using the on-board Warp machine [Pomerleau 89]. The neural net is no longer the limiting factor, and Navlab can now be trained just by driving it on the road for a short time (2 minutes), and can drive up to 8 miles per hour. Early versions of ALVINN used both a low-resolution video image and a laser rangefinder; at present, only the visual input is needed.

1.4. Learning and problem solving architectures

Our objective is to develop computer systems capable of solving problems and learning in a wide range of complex tasks. Integrating learning and problem solving into a single system is a necessary development for AI to make a major contribution to future DoD systems. The heart of that integration is the ability of the system to continually analyze its own experience and make that available for immediate future action. This section describes our continuing work on Prodigy and Soar, two integrated intelligent architectures that are to provide such capabilities. These architectures aim for generality by providing problem-solving and learning mechanisms at a foundational level, in contrast to the traditional expert systems approach of building special-purpose mechanisms for new applications.

1.4.1. Soar

The Soar system is an architecture for general intelligence. Soar incorporates the ability to solve problems in both knowledge-lean and knowledge-intensive situations, to exhibit the full range of appropriate problem-solving methods, and to learn from its experience about all aspects of its operation. Soar has been under development since 1982 and is a maturing system that we have used to conduct research in AI, cognitive psychology, and implementation technology. On the AI side, Soar aims to be an architecture that can be used for the full range of AI applications. On the psychological side, Soar is being used as a basis for a unified theory of human cognition, including an engineering model of the user for use in human-computer interface design (extending work started by Card, Moran and Newell in the 1970s). In terms of implementation technology, there is a substantial effort within the Soar project to explore efficiency issues for production systems, especially on parallel machines.

Research is distributed over three main sites where the primary researchers reside: CMU (Newell), University of Michigan (Laird); and Information Sciences Institute at USC (Rosenbloom). Additional participants are working on Soar at about half a dozen other sites. The various Soar sites form a tightly integrated community, but informal specializations have developed, with Michigan concentrating on using Soar as a robot controller, and USC focused on mainstream AI areas such as abstraction, planning, and the relationship of Soar to connectionism. The research at CMU covered during the term of this contract has focused on specific AI applications as well as the support of general capabilities and implementation support for Soar.

AI Applications

AT CMU, Soar has been applied to several difficult knowledge-intensive areas, taken mostly from engineering-related applications, but also in related areas such as science tutoring and production planning. This work built on previous work at CMU on reimplementing the R1/XCON computer configuration expert system in Soar and at Stanford on medical diagnosis (ongoing work at the Ohio State University continues to explore medical applications of Soar). These multiple foci arise in part from cooperative efforts initiated by domain principals outside the DARPA project (which is our deliberate strategy). The CMU efforts during the term of this contract are briefly described below:

- **Algorithm design**: Based on the detailed analysis of the behavior of human algorithm designers conducted as part of the Designer project, we built several versions of an automatic algorithm design system in Soar. The last version, Designer-Soar [Steier and Newell 88], integrated knowledge about general problem-solving, algorithm design, implementation techniques, and the application domain. The system used several levels of abstraction, generalizes from examples, and learns from experience, transferring knowledge acquired during the design of one algorithm to aid in the design of others. Along the way, we produced a monograph that is the first systematic comparison of published derivations and syntheses of algorithms. The comparison focused on seven algorithms for which multiple derivations were available in the literature: insertion sort, quicksort, Cartesian set product, depth-first search, Schorr-Waite graph marking, N-queens, and convex hull). We also initiated small efforts to extend the framework of Designer-Soar to other areas of programming technology, such as coding, and system design, and data structure design.

- **Chemical engineering**: We built a Soar system to perform simple design of simple chemical separations systems. Separations systems design involves determining the sequence in which splits should be carried out in order to isolate the constituent components of a given feed mixture. To evaluate competing split sequences, this design task requires the extensive use of specialized software such as chemical process simulators. Experienced engineering designers employ heuristics, such as doing the easiest split first, to minimize the computational effort required. The first Soar-based separation systems designer, CPD-Soar (Chemical-Process-Design-Soar), employed such heuristics to design simple systems of cascaded distillation columns. However, a major limitation of this system was that the majority of rules it learned encoded specific numeric values in their conditions, and thus would not transfer to new tasks. A second system, Interval-Soar, was built to better understand the processes an agent should use to be able to generalize intervals from specific quantitative results. The experience from Interval-Soar and CPD-Soar was combined to produce a preliminary design for CPD2-Soar, and hand-simulations of this design showed that useful learning would be obtained. The design of CPD2-Soar also yielded a new result from the standpoint of chemical engineering: tests run on hundreds of sample design problems led to the discovery of a powerful new heuristic for distillation system design that was later shown to outperform every other known heuristic for that class of problems.

- **Civil engineering**: Two systems were built to explore the potential of Soar applications in this domain. The first system implemented the design of a floor system. A floor system is a floor slab covering a typical bay in a building floor, and its design involves choosing the action of the slab (whether it will bend along one or two dimensions due to gravity), the type of material and the type of support. Several types of learning were demonstrated to be possible with careful problem structuring. The second system used Soar to learn tool selection knowledge in an integrated building design environment. The environment contains seven tools that can be sequenced to generate a construction plan for the structural system of a high rise office building, given a set of owner requirements for area, cost, and siting. The task of the Soar system was to learn the proper order in which to invoke the tools, extracting the necessary knowledge from experimentation, a model of the inputs and outputs of each tool, and guidance from an expert user. Several problem solving methods were implemented including forward search and means-ends analysis; learning was shown to reduce the problem solving effort significantly (by as much as two-thirds) on all of these methods.
- **Manufacturing scheduling**: Since the summer of 1988, part of the applications effort has focused on building a prototype scheduling system for the production of replacement automobile windshields. The production process involves cutting and bending the glass and stretching screening, and cutting vinyl. The focus of the system, now in its second version as Merle2-Soar, is the scheduling of the bending of the glass, done in a large oven called a lehr. The schedule must satisfy both hard and soft constraints. Hard constraints reflect the physical realities of the factory, such as lehr capacity, and must be satisfied for the schedule to be feasible. Soft constraints are preferences which should be taken into consideration to increase the quality of the schedule. For example, all other things being equal, it is desirable to interleave large and small jobs to avoid exhausting the workers. Merle2-Soar, which incorporates most of the hard constraints of the factory situation, demonstrated both across-trial and across task transfer in producing schedules.
- **Electrostatics tutoring**: ET-Soar is a Soar-based tutoring system designed to teach students how to use field diagrams to solve problems in electrostatics. The system embodied a framework based on a problem space-based model of the tutoring situation: a state representation is used for the curriculum and student model and tutoring strategies are represented as operators that can be applied to eventually yield a final state in which the student has learned the curriculum. A general Soar-based model of agent-tracking (following the visible actions of an external agent using an internal cognitive model of the task being performed) is used as the basis for student diagnosis via model tracing. To overcome

performance limitations of ET-Soar, a novel experimental technique was used in which a slow computer tutor is replaced with a hidden human operator and the results of that session are fed back to the computer-based tutor (at a slower pace) to provide for complete validation of the tutor. Using this technique, ET-Soar was successfully used to teach actual students the use of electric field diagrams.

General Capabilities

The work on specific Soar applications was complemented by two efforts on providing general capabilities. These efforts both began with the construction in 1987 of a task acquisition system that permitted Soar to acquire new tasks from the environment by learning. This system used Soar's learning mechanisms to acquire new tasks from a user. This system had both a formal language input for describing Soar systems in terms of problem spaces, and an (elementary) capability for natural language input. Each of these input capabilities subsequently developed into a major research thrust at CMU, with about four to six people devoted to them apiece. The first group, RTAQ, focuses on specification of Soar systems, while the second, NL-Soar, focuses on providing Soar with the ability to understand natural language input as it runs. Both efforts are the beginnings of language capabilities that will eventually make it easier for humans to interact with Soar.

The RTAQ (Rapid Task AcQuisition) subgroup of Soar has as a core belief that constructing initial expert system versions in Soar seems to require only identification processes, assignments of known representational schemes, and communication linkages [Yost and Newell 89]. The usual lengthy process of explicitly designing methods is not necessary, because Soar operates as a multiple-problem space system. Thus the expectation is that expert system construction will be facilitated by providing the appropriate language and environment for describing systems at this level. The result at the end of the contract period was the Task Acquisition Language (TAQL). We identified a number of small tasks for testing TAQL, and in a cooperative effort with Digital Equipment Corporation, collected a set of test cases of medium-sized domain-oriented expert system specifications (where the task descriptions are several thousand words long). These were used to drive the development of TAQL, and associated tools such as structured editors and a graphical interface. TAQL also contains capabilities for interfacing to databases and specifying operator control; these constitute extensions to our original conception of the problem space computational model. By now, TAQL has been extensively tested in two rounds of experiments, resulting in system building times on the order of several minutes per production for systems with a few hundred productions. There is a manual for the system, and we have conducted several tutorials on TAQL. TAQL now has about a dozen users within the Soar community, and indeed some of the work described in the previous section, for instance on manufacturing scheduling, is now conducted entirely within TAQL.

Since our original interest in giving Soar a natural language capability was as another vehicle for rapid task acquisition, our original work in language comprehension was done as part of TAQ. Within the course of this contract, however, the natural language effort has become distinct from TAQL and continues to follow its own course. At the basis of the system is the idea of the *comprehension operator*, first introduced in the William James Lectures in 1987. The idea of the comprehension operator is a general one, extending beyond language to vision and the other ways in which we comprehend the environment. With respect to language, however, the comprehension operator brings to bear all the knowledge about a word in a given context to produce data structures in working memory that can be used by later comprehension and by problem solving.

The comprehension operator approach allowed us to build a 1988 version of NL-Soar that greatly improved Soar's natural language capability for simple task instruction. This version of the system was integrated with another Soar system (IR-Soar) having an immediate reasoning ability, that is, the ability to extract implicit information from simple situations in 30 seconds or less (for example, inferring the truth of a discrete statement based on given premises). After expanding the immediate reasoning tasks and studying Soar's performance, we found that Soar's immediate reasoning process is very similar to humans' [Lewis et al. 89, Polk et al. 89]. One difficulty with the natural language portion of the integrated system, however, was that the comprehension operators had to be constructed by hand; hand-coding of operators that integrate multiple knowledge sources has been demonstrated to be an extremely difficult design problem for a number of systems.

As a solution to this difficulty, the final version of NL-Soar constructed during the contract period was restructured to take advantage of Soar's chunking mechanism in order to learn comprehension operators automatically. The system is now organized so that it performs both deliberate processing (the sequential application of syntactic, semantic, and pragmatic knowledge sources) and recognitional processing (the simultaneous, or parallel, application of those knowledge sources). If no comprehension operator exists for the word in the current context, the system proceeds deliberately. That deliberate processing is then captured by chunking so that processing can proceed recognitionally in future, similar circumstances. Thus, the two activities intermix freely and provide a paradigm for Soar to move from deliberate activity to skill. The 1990 version of NL-Soar uses an all-paths, bottom-up parsing algorithm and a chart-like structure containing standard phrase-structure constituents to build the annotated models that represent the meaning of the utterance.

To the major thrusts on rapid task acquisition and natural language comprehension described above should be added a list of smaller (essentially single-person) efforts on general capabilities in Soar. An example that is related to the NL-Soar effort is the use of annotated models as a representation of situations. Another project has tried to understand what allows people to acquire new strategies for solving small tasks such as the Tower of Hanoi puzzle [Ruiz and Newell 89]. Additional even smaller efforts have studied representation shifts, the use of constraint satisfaction, and the basic code used by humans to represent small quantities, so that for example, one can recognize immediately that there are four blocks on the table without counting. Much of this work was used to complete a manuscript for a book based on the William James Lectures in

Psychology delivered at Harvard in the spring of 1987, and again in the fall at CMU. Although focused on human cognition, the book was built entirely around the Soar system and the suitability of its architecture for obtaining intelligent action.

Implementation support

During this period we also made several improvements to Soar, further expanding its capabilities and building a more efficient software technology base.

A major part of this effort was due to the first major revision of Soar's underlying architecture, working to ensure that all learning and performance capabilities were preserved in a transition from Soar4 to Soar5. Soar5 uses a "rolling" state which is continually modified, rather than copying all problem-solving states, as the prior Soar systems did. Although this strategy appears relatively straightforward, in fact it induces profound changes by altering the production system's basic computational model and the nature of working memory, forcing us to use a truth-maintenance system, and having strong interactions with chunking. Despite almost 3 years of development and analysis, new conceptual issues have continually emerged as we have gained experience with the system. These challenges require considerable analysis, some system redesign, and significant reworking of specific Soar systems. Almost all the main projects (some 20 systems at CMU, Michigan and ISI) have now converted completely to Soar5 and are running successfully including learning. We have completed a new draft manual. In addition, we converted TAQL, the problem-space language discussed in the previous section, to produce Soar5 systems.

Another issue related to the implementation of Soar has been the slowdown in the match that sometimes results when Soar's chunking is used. The primary cause of the slowdown are what we have called expensive chunks, single productions that drastically increase the time per step needed to solve problems [Tambe and Newell 88]. After much investigation, we have found partial means to control the phenomena of expensive chunks, allowing Soar to remain faithful to its constant time per step model [Tambe and Rosenbloom 88]. The central notion in the solution is to restrict the expressiveness of the language in which Soar's productions are written. When chunking is applied to the restricted language, we can guarantee that chunking will create only *cheap*, i.e.. nonexpensive chunks. We have convinced ourselves that the drawback of restricting expressiveness is more than adequately compensated for by the resulting gain in efficiency.

The third implementation area explored during the term of this contract was the investigation of advanced software and hardware techniques to increase the efficiency of Soar. Several improvements in the match network implementation, including a conversion from Lisp to **C**, have resulted in significant speedups. But potentially the most significant speedups will come from the use of parallelism. In 1989, we introduced ParaOPS5, a parallel version of OPS5 (a central part of our underlying software). A parallel version of Soar 4 now operates on the Encore multimax (16 processor shared-memory system) and we have obtained the first measurements showing good, but not fully linear, speedups. These implementations have been facilitated by detailed studies of production systems. These studies uncovered features, such as long chains of match tasks that would limit parallelism if existing uniprocessor implementations would be straightforwardly converted for use on multiprocessors. A particularly interesting finding was that we found an increased potential for parallelism in learning systems over nonlearning systems, due to the increased size of the affect set as new productions are added to the network.

Concurrently with these research areas, we are continuing to support Soar as an experimental computer system, now used by perhaps 170 people at a dozen sites. The development of a manual, system support, and training materials for Soar has been a high priority for the project. As part of the support effort, we gave a 4-day tutorial and hands-on workshop at the University of Groningen, The Netherlands, in June 1990. We prepared new documentation and teaching exercises for the event. This requires substantial effort, but pays off in getting Soar systems developed in many different directions (as in discovering the conceptual issues with the new Soar5 system).

1.4.2. Prodigy

Prodigy is a computational architecture designed as a general testbed for research in problem solving, planning, and most centrally, machine learning. This section reviews the objectives of the project, the research methodology, results obtained, and current (new) directions including promising application directions.

The basic motivation

Effective acquisition of large amounts of knowledge has proved to be a bottleneck for the construction of large knowledge-based systems. A promising paradigm entails separating factual knowledge (knowing what) from control knowledge (knowing how and when to apply facts to solve problems). Experience shows that the former knowledge is easier to acquire as it is more readily formalized both in written texts and in the minds of experts. The latter knowledge is often tacit, problematic to convey, and difficult to acquire manually. Recent advances in machine learning, however, have focused precisely in the acquisition of tacit control (strategic) knowledge from experience. In particular, explanation-based learning, analogical case-based reasoning, and automated formulation of abstraction hierarchies are promising techniques for effective acquisition of control knowledge.

In order to exploit these machine learning techniques, however, an integrated architecture is required where the learning component feeds new knowledge to the performance component (the planner and problem solver), and the latter in turn provides feedback as to the effectiveness of new control rules, etc. Prodigy is designed precisely to address this type of learning-performance integration, starting with a powerful problem solver and extending to numerous learning techniques, all sharing the same underlying knowledge representation (a form of typed first-order logic with inheritance), and subject to a uniform control structure.

Research Strategy and its execution

Our research strategy called for a 3-year plan concentrating first on developing of the core planning system, then the various learning modules, then testing on several domain, and then evaluating and disseminating results (including the software) widely within the AI community (academic, government and industrial). The underlying hope was for a stable computational architecture that would enable research into the various machine learning methods to proceed systematically and with clear comparative and evaluative criteria.

The first year of the research focused on the completion of the basic Prodigy problem solving and planning architecture with the appropriate "hooks" (connection points) to the multiple learning mechanisms under parallel (or subsequent) development. Additional activities included initial work on learning modules, documentation of Prodigy 1.0, and developing test domains.

The second year focused on developing and integrating three core machine learning modules: Explanation-Based Learning, Derivational Analogy, and manual knowledge acquisition. Additional activities included testing, evaluation and debugging of the core Prodigy system (including alpha distribution to "friendly" external laboratories, integration of the learning and performance components, and developing more extensive problem suites and domains of application. Initial work was started on Prodigy 3.0, the nonlinear complete planner. The latter activity was unplanned, but developments in general planning methodologies elsewhere made it possible to accelerate our effort and provide a state-of-the art complete nonlinear planner—for the first time ever, one capable of learning from experience.

The third year witnessed some thorough evaluation of the integrated learning and performance components of Prodigy (most notably the linear planner with explanation-based learning), widespread distribution and external feedback of the robust Prodigy 2.0, continued development of the full nonlinear Prodigy 3.0 version, and the addition of new learning methods such as the automated formulation of abstraction hierarchies for hierarchical planning (performing bottleneck and contention analysis to define the most effective abstractions). Multiple domains were defined and used to test the effectiveness of the learning mechanisms bearing out their generality across domains as diverse as robotic path planning, factory-floor machining and scheduling, matrix algebra computations and computer configuration.

Most of the work followed our original strategy and implementation plan, with some additional unplanned directions such as nonlinear planning (mentioned above), testing on 10 different domains instead of just the four-to-six initially envisioned, and distributing to more external sites, as there was a demand for the software, however experimental.

The one aspect of our research strategy not thoroughly completed involved scaling up experiments. These became the first priority in the following contract, with very positive initial results.

The most gratifying aspect of the project is the incremental nature of the results. Improvements in the performance system are immediately available to all the learning modules, and the effects of each type of leaning can be measured quantitatively (e.g. as time/space improvements in problem solving performance) allowing for direct comparisons. In essence Prodigy already serves as the unified "plugboard" to investigate learning methods in the context of a performance engine. Testing is an integral part of development and points the way to the next functional improvement.

Research Results and Applications

The results of the various learning methods developed in Prodigy are reported at length in the published literature (see bibliography). Here we summarize the primary achievements in an integrated manner.

Performance Engine

The central performance engine in Prodigy is a domain-independent planning system, of which there are three interchangeable versions:

- Prodigy 1.0 is a linear operator-based planner that operates with a default means-ends strategy, modifiable by control rules that focus search (deciding which operator to apply, which goal to pursue, which objects to select, etc.) These control rules can be hand-coded or acquired via the learning mechanisms such as EBL (see below).
- Prodigy 2.0 is a hierarchical planner that uses 1.0 as its basic engine, but can plan in abstract spaces (to produce skeletal plans), and the refines these plans gradually reintroducing details until a full plan is achieved (or proved impossible). Abstraction is useful when the correct details are suppressed to reduce search and are later reintroduced without significant alterations in the skeletal plan. Much more complex problems can be addressed in this manner.
- Prodigy 3.0 is a nonlinear "complete" planner capable of interleaving goals and subplans to cope with interactions in the most efficient manner. Unlike the linear planners, 3.0 can follow any control discipline best suited for the task at hand.

The software for Prodigy 2.0 has been sufficiently documented and tested on site, as well as at various other laboratories, so that it can be pronounced "robust." The same is not yet true for 3.0, but it is similarly scheduled for external release and testing in subsequent funding phases. Several sites are actively using Prodigy 2.0 as the basis for their planning and learning R&D projects. Copies have been distributed to many sites including Naval Research Laboratories, GTE laboratories, and several universities.

Learning

Many learning modules have been built and tested on the Prodigy performance engines (above). These include:

- ***Explanation-Based Learning*** (EBL): EBL traces the execution path of past problem solving events to extract the critical decision points (those where incorrect decisions produce substandard planning behavior, wasted time or resources, etc.) in order to generate new control rules so that the next time an equivalent decision presents itself, a more informed (and therefore more correct) choice will be learned. Of course, there is some overhead in learning and applying more control rules. Therefore, utility analysis is performed to determine which control rules to retain (those whose effectiveness outweighs their application cost). Note that an integrated learning-performance is required in order to automate the feedback into the utility metric. EBL has demonstrated performance improvement factors ranging from 2X to 6X in several domains: robotic path planning, extended STRIPS, and so on.

- ***Abstraction***: Prodigy is capable of learning abstraction hierarchies (for 2.0 version hierarchical planning) by analyzing the domain definition (operators, inference rules, relations and objects) in order to identify critical paths and solve problems by first addressing the most important and contentious issues and later reintroducing other details progressively. Multi-layered abstraction hierarchies were produced automatically for domains as simple as "tower of Hanoi" and as complex as machine-shop process planning and scheduling. Performance improvements comparable to those of EBL were obtained (2X to 10X), and future research will address the integration of both methods to determine whether their combination is capable of higher (additive or multiplicative) performance improvements.
- ***Derivational Analogy***: Unlike EBL and Abstraction, DA does not require a full domain definition to learning from experience. In essence, DA stores past solution paths to problems in a large case memory indexed by goal, initial state, etc., and retrieves these solutions when encountering new similar problems in future situations. Retrieved solutions are "replayed"; that is, the same lines of reasoning are recreated, modifying the final plan as necessary to accommodate differences between new and old problems. Although DA can require a large case memory and pay an overhead price for indexing and retrieval, experiments have shown that improvements in performance of up to 20X have been demonstrated in nonlinear planning tasks. Future research will address the scaling up issue and the integration of DA with EBL and Abstraction to see if is possible to obtained the combine best behavior from all three.

All three of these learning mechanisms have been implemented and tested in a number of domains. Other learning mechanisms have been explored but not found as effective. New ones, however, will continue to be explored in a systematic manner thanks to the common performance engine, domain definitions that allow rapid testing and evaluation, and easy integration.

1.4.3. Comparison of Soar and Prodigy

Soar and Prodigy are both exploring mechanisms, built in at the architectural level, that facilitate integrated problem-solving and learning. An important similarity between the two architectures is that each is built around a relatively simple general problem solver whose performance improves incrementally through the acquisition of factual and control knowledge for each domain. The differences arise from the different goals and assumptions of each project:

- **Access to long-term memory**: All the knowledge used or acquired in any Prodigy module is open for inspection and use by every other module, a discipline made possible by the use of a uniform logic-based representation of all control and factual knowledge. In contrast, once a Soar chunk is formed, its contents are not open to inspection or modification. Prodigy's structure is based on the desire for representational transparency, while the structure in Soar is based on psychological plausibility and efficiency considerations.
- **Variety in learning vs. variety in problem solving**: A number of learning strategies are distinct architectural mechanisms in Prodigy: explanation-based learning, analogy, abstraction, experimentation, static analysis, tutoring, etc. Soar has a single architecturally-defined learning mechanism, chunking, but can exhibit the functionality of the different learning mechanisms if it can use the appropriate knowledge during problem solving. Similarly, Prodigy embodies a commitment to a single problem-solving method, backwards chaining, while Soar has been provided with default knowledge to use a variety of the classical weak problem-solving methods, including means-ends analysis, lookahead search, hill-climbing, etc.
- **Deliberative vs. reflexive learning**: Prodigy acquires new knowledge only when it believes that knowledge will be useful; learning is a deliberate meta-reasoning process. Soar, on the other hand, learns all the time, based on the assumption that the future utility of learned knowledge cannot be predicted in a general way.

1.5. Automated Feature Analysis

Since the fall of 1987, the Digital Mapping Laboratory has been investigating knowledge-intensive techniques for the detailed analysis of remotely-sensed imagery. Our strategy has been to develop rule-based scene interpretation systems for airports and urban areas. This work has resulted in the design and implementation of several rule-based image analysis systems and supporting work in knowledge acquisition and performance analysis tools.

We expect to establish specific performance levels for automated feature extraction techniques that can be used in emerging modernization programs within DoD and the intelligence community. While there are high expectations for the utility of knowledge-based systems as intelligent aids for imagery analysts, concrete results and prototype implementations are still in their infancy.

We have focused our efforts in two broad areas. We will begin by describing our advances in the area of knowledge acquisition, followed by our accomplishments in applying task-level parallelism to rule-based systems.

1.5.1. Acquisition of Spatial and Functional Knowledge

For rule-based systems to be at all practical, it is important that there are tools available to create, maintain, and evaluate a system's knowledge base. By the spring of 1987, we had already embarked on an effort to aid in the interactive accumulation of spatial knowledge for SPAM. This included:

- Pioneering the development of techniques for the automatic compilation of spatial and structural constraints into OPS5 productions, which can be directly executed by the SPAM interpretation system.
- The integration of automated performance evaluation tools in order to study the effect of various types of knowledge in the quality of the overall scene interpretation. This work relies on a database of human generated ground truth interpretations.

This work made it possible to extend the scope of the SPAM system from its original task of analyzing airport scenes to also be able to interpret suburban house scenes.

From the Fall of 1987 through the Spring of 1988, a suite of tools was developed for interactive knowledge acquisition. This collection includes tools for result evaluation, as well as for knowledge acquisition and compilation.

- RULEGEN, a compiler that converts knowledge represented as schemata into OPS5 productions.
- PHOTOGRAM, a photogrammetric measurement module.
- SPAMEVALUATE, an interactive tool for graphically "navigating" through and evaluating the results generated by a SPAM run. This includes tools for acquiring knowledge for the first (RTF) and second (LCC) phases of SPAM.
- SPATS, an automated performance analysis system which uses ground truth scene segmentations to analyze scene interpretations produced by SPAM, our knowledge-intensive scene interpretation architecture.

Tool refinement and development has continued throughout the duration of this contract and beyond. For example, we have developed and tested automated (unsupervised) methods for knowledge acquisition. Our current expectation is that such methods, though marginally useful by themselves, will be useful as methods of suggesting appropriate rules to the user of an interactive acquisition system.

We continue to investigate techniques for representing functional and spatial constraints, typically used by cartographers and imagery analysts, in an automated knowledge acquisition system. This work has provided, and will provide, important insights into the feasibility of automating the process by which scene interpretation systems acquire new tasks and become more proficient in old ones. Therefore, this continues to be a major emphasis in our research.

1.5.2. Parallel Execution of Rule-Based Systems

Large production systems (rule-based systems) suffer from extremely slow execution that limits their utility in practical as well as research applications. Efforts focusing on match (knowledge-search) parallelism have not yielded sufficient speedups.

In the Fall of 1988, with our knowledge acquisition tool suite operational, we began to investigate the application of task-level parallelism to high-level vision. We used the SPAM interpretation system, comprising over 700 OPS5 production rules, as the basis of our investigations to:

- Measure how task decompositions yield effective parallelism;
- Study scheduling issues due to task granularity;
- Study interactions between knowledge-search parallelism and task-level parallelism.

Our initial investigations focused on a SPAM processing phase that used task knowledge to apply and propagate local spatial constraints between object hypotheses in a scene. We used the airport interpretation task, which applies general and structural constraints to airport layout. The interpretation phase, local consistency check (LCC), takes between 40 and 140 hours of CPU time on a VAX-11/785.

After some initial attempts revealed difficulties in dealing with a Lisp-based implementation, and to capitalize on the best existing technology, we reimplemented SPAM in ParaOPS5, a **C**-based, optimized, parallel implementation of OPS5. This provided us with an initial 10-20 fold speed-up over the original Lisp-based SPAM. This move has also enabled us to port SPAM to several different hardware platforms, including including the Encore Multimax, the DEC 3100, and the Sun SPARCstation.

We can decompose SPAM's LCC phase computation into independent subtasks at different granularities, and our intent was to attempt to discover the optimal granularity. We experimented with two different decompositions, level 3 and level 2. Level 3 has 50-200 independent subtasks, each executing in approximately 5 seconds. Level 2 has 400-1000 available subtasks, each executing in approximately 1.5 seconds.

We performed experiments on the Encore Multimax, using three large data sets representing three different airports. The results are reported in Section 1.1.2 of this report, as part of our cooperative work with Task Level Parallelism.

Our local computing environment has two Encore Multimax machines, each with 16 processors. In the Spring of 1988, SCS researchers made available the *shared memory server*, providing a virtual 32-processor shared memory machine. At this time, we began investigating the *shared virtual memory* system, which provides a virtual address space among all processors in a loosely-coupled multiprocessor. Introducing shared virtual memory into the SPAM/PSM system was more complex than our initial expectations. In this configuration, the programmer has to be more sensitive to allocating data structures to pages in order to avoid contention. Though many problems with this configuration remain, we were able to show that significant speedups are possible (15-

fold speedup using 22 processors) and that the overheads associated with using the shared memory server are not excessive.

Our latest work in the area of task-level parallelism (TLP) has emphasized making it a usable tool for SPAM researchers. We continue to make performance improvements to the baseline CParaOPS5 system, such as improving memory efficiency. We have also removed some of the experimental features of the TLP implementation, such as the throwing away of computed results when the execution terminated. In the process, we have identified and measured the contention on several shared resources and, in some cases, have been successful in removing the contention. Though the system has produced good speed-ups (12.5-fold using 14 processors), we believe there are still improvements that can be made. This continues to be a topic of investigation.

1.6. Bibliography

[Acharya and Tambe 89a]
Acharya, A., and M. Tambe.
Production systems on message passing computers: Simulation results and analysis.
In *Proceedings of the International Conference on Parallel Processing*, pages 246-254. August, 1989.

[Acharya and Tambe 89b]
Acharya, A., and M. Tambe.
Efficient implementations of production systems.
VIVEK: A quarterly in artificial intelligence 2(1), 1989.

[Anantharaman et al. 88]
Anantharaman, T., M. Campbell, and F. Hsu.
Singular extensions: adding selectivity to brute force searching.
In *Proceedings of the 1988 AAAI Spring Symposium on Computer Game Playing* . American Association of Artificial Intelligence, March, 1988.

[Berliner and Ebeling 88]
Berliner, H. and C. Ebeling.
SUPREM: architecture based on pattern knowledge and search .
In *Proceedings of the 3rd International Conference on Supercomputing* . IEEE, May, 1988.

[Campbell 88]
Campbell, M.
Chunking as an abstraction mechanism.
Technical Report CMU-CS-88-116, Computer Science Department, Carnegie Mellon University, February, 1988.

[Carbonell and Gil 87]
Carbonell, J.G. and Y. Gil.
Learning by experimentation: the operator method.
Technical Report 87-167, Computer Science Department, Carnegie Mellon University, September, 1987.

[Carbonell and Veloso 88]
Carbonell, J.G. and M. Veloso.
Integrating derivational analogy into a general problem solving architecture.
In *Proceedings of the Case-Based Reasoning Workshop*. DARPA, May, 1988.

[Carbonell et al. 89]
Carbonell, J.G., C.A. Knoblock, and S. Minton.
PRODIGY: An integrated architecture for planning and learning.
Technical Report CMU-CS-89-189, School of Computer Science, Carnegie Mellon University, October, 1989.

[Chiles 89] Chiles, B.
CMU Common Lisp user's manual: Mach/IBM RT PC edition.
Technical Report CMU-CS-89-132-R1, School of Computer Science, Carnegie Mellon University, November, 1989.
Revision of CMU-CS-89-132.

[Chiles and MacLachlan 89a]
Chiles, B., and R.A. MacLachlan.
Hemlock user's manual.
Technical Report CMU-CS-89-133-R1, School of Computer Science, Carnegie Mellon University, November, 1989.
Revision of CMU-CS-89-133.

[Chiles and MacLachlan 89b]
Chiles, B., and R.A. MacLachlan.
Hemlock command implementor's manual.
Technical Report CMU-CS-89-134-R1, School of Computer Science, Carnegie Mellon University, November, 1989.
Revision of CMU-CS-89-134.

[Doyle 88a] Doyle, J.
On universal theories of defaults.
Technical Report CMU-CS-88-111, Computer Science Department, Carnegie Mellon University, March, 1988.

[Doyle 88b] Doyle, J.
On rationality and learning.
Technical Report CMU-CS-88-122, Computer Science Department, Carnegie Mellon University, March, 1988.

[Doyle 88c] Doyle, J.
Similarity, conservatism, and rationality.
Technical Report CMU-CS-88-123, Computer Science Department, Carnegie Mellon University, March, 1988.

[Doyle 88d] Doyle, J.
Artificial intelligence and rational self-government.
Technical Report CMU-CS-88-124, Computer Science Department, Carnegie Mellon University, March, 1988.

[Doyle 88e] Doyle, J.
Implicit knowledge and rational representation.
Technical Report CMU-CS-88-134, Computer Science Department, Carnegie Mellon University, April, 1988.

[Fahlman 87a] Fahlman, S.E.
Common Lisp.
Annual Review of Computer Science 2, 1987.

[Fahlman 87b] Fahlman, S.E.
Connectionist architectures for AI: Problems and prospects.
In *Workshop on Concepts and Characteristics of Knowledge-Based Systems*. International Federation of Information Processing Societies, November, 1987.

[Fahlman 88] Fahlman, S.E.
Faster-learning variations on back-propagation: an empirical study.
In *Proceedings of the 1988 Connectionist Models Summer School*. 1988.
Also available as technical report CMU-CS-88-162.

[Fahlman and Lebiere 90]
Fahlman, S.E. and C. Lebiere.
The Cascade-Correlation learning architecture.
In Touretzky, D.S. (editor), *Advances in Neural Information Processing Systems*. Morgan-Kaufman, Los Altos, CA, 1990.
Also available as technical report CMU-CS-90-100.

[Fahlman and McDonald 90]
Fahlman, S.E. and D.B. McDonald.
Design considerations for CMU Common Lisp.
Advanced Language Implementation Techniques.
In Lee, P.,
Springer-Verlag, 1990.

[Goetsch and Campbell 88]
Goetsch, G., and M. Campbell.
Experiments with the null move heuristic in chess.
In *Proceedings of the 1988 AAAI Spring Symposium on Computer Game Playing*. American Association of Artificial Intelligence, March, 1988.

[Golding et al. 87] Golding, A., P. S. Rosenbloom, and J. E. Laird.
Learning general search control from outside guidance.
In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*. August, 1987.

[Gupta and Tambe 88]
Gupta, A. and M. Tambe.
Suitability of message passing computers for implementing production systems.
In *Proceedings of the National Conference on Artificial Intelligence*. American Association of Artificial Intelligence, 1988.

[Gupta and Tucker 88]
Gupta, A., and A. Tucker.
Exploiting variable grain parallelism at runtime.
In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programing: Experience with applications, languages and systems*. ACM/SIGPLAN, July, 1988.

[Gupta et al. 87] Gupta, A., C.L. Forgy, D. Kalp, A. Newell, and M. Tambe.
Results of parallel implementation of OPS5 on the Encore multiprocessor.
Technical Report CMU-CS-87-146, Computer Science Department, Carnegie Mellon University, August, 1987.

[Gupta et al. 88] Gupta, A., C. L. Forgy, D. Kalp, A. Newell, and M. Tambe.
Parallel OPS5 on the Encore Multimax.
In *Proceedings of the International Conference on Parallel Processing*. August, 1988.

[Gupta et al. 89] Gupta, A., M. Tambe, D. Kalp, C.L. Forgy, and A. Newell.
Parallel implementation of OPS5 on the Encore Multiprocessor: Results and analysis.
International Journal of Parallel Programming 17(2), 1989.

[Harvey et al. 89] Harvey, W., D. Kalp, M. Tambe, D. McKeown, and A. Newell.
Measuring the effectiveness of task-level parallelism for high-level vision.
Technical Report CMU-CS-89-125, School of Computer Science, Carnegie Mellon University, March, 1989.

[Harvey et al. 90] Harvey, W., D. Kalp, M. Tambe, D. McKeown, and A. Newell.
The Effectiveness of Task-Level Parallelism for High-Level Vision.
In *Proceedings of the ACM/SIGPLAN Symposium on Principles and Practices of Parallel Programming*. ACM/SIGPLAN, March, 1990.

[Heydon 90] Heydon, A.
An introduction to circuit complexity and a guide to Hastad's proof.
Technical Report CMU-CS-90-141, School of Computer Science, Carnegie Mellon University, June, 1990.

[Horowitz 88] Horowitz, M.L.
Automatically achieving elasticity in the implementation of programming languages.
PhD thesis, Computer Science Department, Carnegie Mellon University, January, 1988.

[Hsu et al. 88] Hsu, W.L., A. Prietula, and D. M. Steier.
Merl-Soar: Applying Soar to scheduling.
In *Proceedings of the Workshop on Artificial Intelligence Simulation, The National Conference on Artificial Intelligence*. August, 1988.

[Hsu et al. 89a] Hsu, W., M. Prietula, and D. Steier.
Merl-Soar: Scheduling within a general architecture for intelligence.
In *Proceedings of the Third International Conference on Expert Systems and the Leading Edge in Production and Operations Management*, pages 467-481. May, 1989.

[Hsu et al. 89b] Hsu, W.L., A. Newell, A. Prietula, and D. M. Steier.
Sharing scheduling knowledge between intelligent agents-Extended abstract.
In *Proceedings of the AAAI-SIGMAN Workshop on Manufacturing Scheduling-Eleventh International Joint Conference on Artificial Intelligence*. AAAI, August, 1989.

[Irvin and McKeown 88]
Irvin, R.B., and D.M. McKeown.
Methods for exploiting the relationship between buildings and their shadows in aerial imagery.
Technical Report CMU-CS-88-200, Computer Science Department, Carnegie Mellon University, December, 1988.

[Iwasaki 88] Iwasaki, Y.
Model based reasoning of device behavior with causal ordering.
PhD thesis, Computer Science Department, Carnegie Mellon University, August, 1988.

[Kalp et al. 88] Kalp, D., M. Tambe, A. Gupta, C. Forgy, A. Newell, A. Acharya, B. Milnes, and K. Swedlow.
Parallel OPS5 user's manual.
Technical Report CMU-CS-88-187, Computer Science Department, Carnegie Mellon University, November, 1988.

[Kulkarni 88] Kulkarni, D.S.
The process of scientific research: The strategy of experimentation.
PhD thesis, Computer Science Department, Carnegie Mellon University, December, 1988.

[Kulkarni and Simon 88]
Kulkarni, D. and H. Simon.
Processes of scientific discovery: the strategy of experimentation.
Cognitive Science 12(2):139-175, April, 1988.

[Kuokka 90] Kuokka, D.R.
The deliberative integration of planning, execution, and learning.
PhD thesis, School of Computer Science, Carnegie Mellon University, May, 1990.
Also available as technical report CMU-CS-90-135.

[Laird et al. 87] Laird, J.E., A. Newell, and P.S. Rosenbloom.
Soar: An architecture for general intelligence.
Artificial Intelligence 33:1-64, 1987.
Also available as technical report CMU-CS-86-171.

[Lehman 89] Lehman, J.F.
Adaptive parsing: self-extending natural language interfaces.
Technical Report CMU-CS-89-191, School of Computer Science, Carnegie Mellon University, August, 1989.

[Lewis et al. 89] Lewis, R.L., A. Newell, and T.A. Polk.
Toward a Soar theory of taking instructions for immediate reasoning tasks.
In *Proceedings of the Annual Conference of the Cognitive Science Society*, pages 514-521. August, 1989.

[Lewis et al. 90] Lewis, R.L., S. B. Huffman, B.J. John, J.E. Laird, J.F. Lehman, A. Newell, P.S. Rosenbloom, T. Simon, and S.G. Tessler.
Soar as a Unified Theory of Cognition: Spring 1990.
In *In Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*. Cognitive Science Society, July, 1990.

[MacLachlan and Chiles 89a]
MacLachlan, R.A., and B. Chiles.
Hemlock user's manual.
Technical Report CMU-CS-89-133, School of Computer Science, Carnegie Mellon University, April, 1989.

[MacLachlan and Chiles 89b]
MacLachlan, R.A., and B. Chiles.
Hemlock command implementor's manual.
Technical Report CMU-CS-89-134, School of Computer Science, Carnegie Mellon University, April, 1989.

[McDonald 89] McDonald, D.B.
CMU Common Lisp User's Manual: Mach/IBM RT PC Edition.
Technical Report CMU-CS-89-132, School of Computer Science, Carnegie Mellon University, April, 1989.

[McKeown 88] McKeown, D.M.
Building knowledge-based systems for detecting man-made structures from remotely sensed imagery.
In *Philosophical Transactions of the Royal Society of London, Vol. 324*, pages 423-435. Royal Society of London, 1988.

[Minton 88a] Minton, S.
Quantitative results concerning the utility of explanation-based learning.
In *Proceedings of the National Conference on Artificial Intelligence*. American Association of Artificial Intelligence, 1988.

[Minton 88b] Minton, S.
EBL and weakest preconditions.
In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning*. American Association of Artificial Intelligence , March, 1988.

[Minton 88c] Minton, S.
Learning effective search control knowledge: an explanation-based approach.
Technical Report CMU-CS-88-133, Computer Science Department, Carnegie Mellon University, March, 1988.

[Minton and Carbonell 87]
Minton, S.N. and J. Carbonell.
Strategies for learning search control rules: An explanation-based approach.
In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*. 1987.

[Minton et al. 87] Minton, S.N., J.C. Carbonell, O. Etzioni, C.A. Knoblock, and D.R. Kuokka.
Acquiring effective search control rules: Explanation-based learning in the Prodigy system.
In *Proceedings of the 4th Conference on Machine Learning*. 1987.

[Minton et al. 89a] Minton, S., J. Carbonell, C. Knoblock, D. Kuokka, and O. Etzioni.
Explanation-based learning: A problem solving perspective.
Technical Report CMU-CS-89-103, School of Computer Science, Carnegie Mellon University, January, 1989.

[Minton et al. 89b] Minton, S., C.A. Knoblock, D.R. Kuokka, Y. Gil, R.L. Joseph, and J.G. Carbonell.
Prodigy 2.0: The manual and tutorial.
Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, May, 1989.

[Minton et. al. 87] Minton, S., J. Carbonell, C. Knoblock, and D. Kuokka.
The PRODIGY system: an integrated architecture for planning and analytical learning.
Machine Learning Journal , 1987.

[Modi et al. 90] Modi, A.K., D. M. Steier, and A. W. Westerberg.
Learning to use Approximations and abstractions in the design of chemical processes.
In *Proceedings of the AAAI Workshop on Automatic Generation of Approximation and Abstractions*. AAAI, July, 1990.

[Nayak, et al. 88] Nayak, P., A. Gupta, and P. S. Rosenbloom.
Comparison of the Rete and Treat production matchers for Soar.
In *Proceedings of the National Conference on Artificial Intelligence*. American Association of Artificial Intelligence, August, 1988.

[Newell 87] Newell, A.
Unified theories of cognition.
The William James lectures .
Harvard University, 1987.

[Newell 89a] Newell, A.
Unified theories of cognition.
Harvard University Press, Cambridge, Massachusetts, 1989.

[Newell 89b] Newell, A.
Putting it all together.
Complex Information Processing: The impact of Herbert A. Simon.
In Klahr and Kotovsky,
Klahr and Kotovsky, 1989.

[Newell 89c] Newell, A.
The quest for architectures for integrated intelligent systems.
In *Proceedings on Innovative Approaches to Planning, Scheduling and Control*. August, 1989.

[Newell et al. 89] Newell, A., P. S. Rosenbloom, and J. E. Laird.
Foundations of Cognitive Science.
In Posner,
Bradford Brooks/MIT Press, 1989.

[Perlant and McKeown 89]
Perlant, F.P., and D.M. McKeown.
Scene registration in aerial image analysis.
Technical Report CMU-CS-89-127, School of Computer Science, Carnegie Mellon University, March, 1989.

[Polk and Newell 88]
Polk, T. and A. Newell.
Modeling human syllogistic reasoning in Soar.
In *Proceedings of the Cognitive Science Annual Conference* . Cognitive Science Society, 1988.

[Polk et al. 89] Polk, T.A., A. Newell, and R.L. Lewis.
Toward a unified theory of immediate reasoning in Soar.
In *Proceedings of the Annual Conference of the Cognitive Science Society*, pages 506-513. August, 1989.

[Pomerleau 89] Pomerleau, D.A.
ALVINN: An autonomous land vehicle in a neural network.
Technical Report CMU-CS-89-107, School of Computer Science, Carnegie Mellon University, January, 1989.

[Pomerleau et al. 88]
Pomerleau, D.A., G.L Gusciora, D.S. Touretzky, and H.T. Kung.
Simulating neural networks at Warp speed: how we got 17 million connections per second.
In *Proceedings of the IEEE International Conference on Neural Networks*. IEEE, July, 1988.

[Printz and Servan-Schreiber 90]
Printz, H. and D. Servan-Schreiber.
Foundations of a computational theory of catecholamine effects.
Technical Report CMU-CS-90-105, School of Computer Science, Carnegie Mellon University, May, 1990.

[Reich and Fenves 88]
Reich, Y. and S.J. Fenves.
Floor-System design in Soar: a case study of learning to learn.
Technical Report EDRC-12-26-88, Carnegie Mellon Engineering Design Research Center, December, 1988.

[Reich and Fenves 89]
Reich, Y., and S. Fenves.
Integration of Generic Learning Tasks.
Technical Report EDRC-12-28-89, Carnegie-Mellon University, December, 1989.

[Rosenbloom et al. 87]
Rosenbloom, P.S., J.E. Laird, and A. Newell.
Knowledge-level learning in Soar.
In *Proceedings of the American Association of Artificial Intelligence - 1987*. AAAI, August, 1987.

[Rosenbloom et al. 88a]
Rosenbloom, P.S., J.E. Laird, and A. Newell.
Meta-levels in Soar.
In Maes, P. and D. Nardi (editors), *Meta-Level Architectures and Reflection*. North Holland, 1988.

[Rosenbloom et al. 88b]
Rosenbloom, P.S., J. Laird, and A. Newell.
A preliminary analysis of the Soar architecture as a basis for general intelligence.
Workshop on Foundations of Artificial Intelligence .
In Kirsch, D. and C. Hewitt,
MIT Press, 1988.

[Rosenbloom et al. 88c]
Rosenbloom, P.S., J. Laird, and A. Newell.
The chunking of skill and knowledge.
Working Models of Human Perception .
In Bouma, H. and B.A. Elsendoorn,
Academic Press, 1988.

[Rosenbloom et al. 89a]
Rosenbloom, P.S., A. Newell, and J.E. Laird.
Towards the knowledge level in Soar: The role of the architecture in the use of knowledge.
In VanLehn, K. (editor), *Architectures for Intelligence*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[Rosenbloom et al. 89b]
Rosenbloom, P.S., J.E. Laird, A. Newell, and R. McCarl.
A preliminary analysis of the Soar architecture as a basis for general intelligence.
In Kirsh, D., and Hewitt, C. (editors), *Proceedings of the Workshop on Foundations of Artificial Intelligence*. MIT Press, Cambridge, Massachusetts, 1989.

[Ruiz and Newell 89]
Ruiz, D., and A. Newell.
Tower-noticing triggers strategy-change in the Tower of Hanoi: A Soar model.
In *Proceedings of the Annual Conference of the Cognitive Science Society*, pages 522-529. August, 1989.

[Steier 87] Steier, D.M.
Cypress-Soar: A case study in search and learning in algorithm design.
In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*. AAAI, August, 1987.

[Steier 89] Steier, D.
Automating algorithm design within a general architecture for intelligence.
Technical Report CMU-CS-89-128, School of Computer Science, Carnegie Mellon University, April, 1989.

[Steier and Newell 88]
Steier, D. and A. Newell.
Integrating multiple sources of knowledge into Designer-Soar, an automatic algorithm designer.
In *Proceedings of the National Conference on Artificial Intelligence*. American Association of Artificial Intelligence, 1988.

[Steier et al. 87] Steier, D.E., J.E. Laird, A. Newell, P.S. Rosenbloom, R.A. Flynn, A. Golding, T.A. Polk, O.G. Shivers, A. Unruh, and G.R. Yost.
Varieties of learning in Soar: 1987.
In *Proceedings of the Fourth International Workshop on Machine Learning*. Morgan-Kaufman, 1987.

[Tambe 90] Tambe, M.
Investigating Alternative Production System Formulations.
PhD thesis, Computer Science Department, Carnegie Mellon University, February, 1990.

[Tambe and Acharya 89]
Tambe, M., and A. Acharya.
Parallel implementations of production systems.
VIVEK: A quarterly in artificial intelligence 2(2), 1989.

[Tambe and Newell 88]
Tambe, M. and A. Newell.
Some chunks are expensive.
In *Proceedings of the Fifth International Conference on Machine Learning*. Cognitive Science and Machine Intelligence Lab, University of Michigan, June, 1988.
Also available as CMU technical report CMU-CS-88-103.

[Tambe and Rosenbloom 88]
Tambe, M. and P. Rosenbloom.
Eliminating expensive chunks.
Technical Report CMU-CS-88-189, Computer Science Department, Carnegie Mellon University, November, 1988.

[Tambe and Rosenbloom 89]
Tambe, M., and P. Rosenbloom.
Eliminating expensive chunks by restricting expressiveness.
In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 731-737. August, 1989.

[Tambe et al. 88] Tambe, M., D. Kalp, A. Gupta, C. Forgy, B. Milnes, and A. Newell.
Soar/PSM-E: investigating match parallelism in a learning production system.
In *Proceedings of the ACM Conference on Parallel Processing*. ACM, 1988.

[Tambe et al. 89] Tambe, M., A. Acharya, and A. Gupta.
Implementation of production systems on message passing computers: Techniques, simulation results and analysis.
Technical Report CMU-CS-89-129, School of Computer Science, Carnegie Mellon University, April, 1989.

[VanLehn 87] VanLehn, K.
Learning one subprocedure per lesson.
Artificial Intelligence 31:1-40, 1987.

[VanLehn and Ball 87]
VanLehn, K. and W. Ball.
A version space approach to learning grammars.
Machine Learning 2(1):39-74, 1987.

[Vicinanza and Prietula 89]
Vicinanza, S., and A. Prietula.
A computational model of musical creativity.
In *(AI and Music Workshop, Eleventh International Joint Conference on Artificial Intelligence)*. AAAI, August, 1989.

[Witbrock and Zagha 89]
Witbrock, M. and M. Zagha.
An implementation of back-propagation learning on GF11, a large SIMD parallel computer.
Technical Report CMU-CS-89-208, School of Computer Science, Carnegie Mellon University, December, 1989.

[Yost and Newell 89]
Yost, G.R., and A. Newell.
A problem space approach to expert system specification.
In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 621-627. August, 1989.

[Zlotnick 88a]
Zlotnick, A.
A discrete scale-space representation.
Technical Report CMU-CS-88-156, Computer Science Department, Carnegie Mellon University, June, 1988.

[Zlotnick 88b]
Zlotnick, A.
Locating corners in noisy curves by delineating imperfect sequences.
Technical Report CMU-CS-88-199, Computer Science Department, Carnegie Mellon University, December, 1988.

[Zlotnick and Carnine 88]
Zlotnick, A. and P. Carnine.
Road finding for road network extraction.
Technical Report CMU-CS-88-157, Computer Science Department, Carnegie Mellon University, June, 1988.

2. RESEARCH IN IMAGE UNDERSTANDING

Image understanding is a technology with applications in diverse tasks, including aerial photointerpretation, automated manufacturing, and robot vehicle guidance. In the past, efforts at image understanding have largely relied on heuristics that use over-simplified models of image and scene. This approach typically produces ‘‘brittle’’ systems with limited scope and capability. Our research aims at making breakthroughs in all aspects of the problem by introducing more sophisticated models of object properties, sensor data, and the imaging process, and by showing how to incorporate these models into demonstrable systems. We concentrate in six areas:

- Understanding image color and texture
- Extracting shape and reflectance
- Determining visual depth and camera motion
- Developing a framework for model-based computer vision
- Acquiring 3-D recognition algorithms
- Fast rangefinding by analog VLSI

2.1. Understanding Color and Texture

Low-level vision systems can succeed only when they adequately account for complex features such as color and texture. Oversimplified models have, in the past, led to programs that cannot deal with the complexity of real images. Our research in color and texture understanding aims to develop more realistic models that allow the representation, analysis, and, hence, ‘‘understanding’’ of complex image effects.

2.1.1. Understanding color

Color offers a rich source of information for analyzing images. However, previous color analysis efforts assumed random color variations and so employed simple statistical approaches for modeling color. Such methods failed to extract such basic information as object boundaries. We have been developing an approach to understanding color based on modeling the physical processes that produce images. This goal requires modeling the illumination of a scene, the way scene objects reflect light, and the way in which the color camera records the image. These models relate characteristics of scene and camera to the resulting color properties of the image. Our model allows us to develop methods that can extract, from image data, important information about the scene.

One of our primary thrusts in this area is modeling the physical laws of reflection and how they determine color in an image. In this work, we previously developed a Dichromatic Reflection Model that describes the physical cause of *gloss*, or highlights, and *object color*, which is the characteristic color of an object. For many dielectric materials such as paint and plastic, these reflections have different colors. The color difference can be described mathematically by applying the laws of color physics. Then, by using the equations that describe how color images are formed, we obtain a model that relates the light reflected by this type of object to the distribution of color data in the image. From this model we derive an algorithm to analyze color images. In 1987, we successfully used this algorithm on actual color photographs of plastic objects. In these demonstrations, we produce two images: One, with the highlights removed, shows just the object's body color; the other contains only the highlights. This result is of considerable importance in computer vision because these resulting images have a much closer correspondence to the underlying geometry of the scene than the original color image. For example, in analyzing surface smoothness and defects, the image of highlights shows these features plainly, whereas they are quite subtle in the original photograph.

In addition to modeling reflection, we developed several methods for modeling the imaging process. Many vision algorithms assume an ideal camera that obeys simple imaging equations. Unfortunately, such an instrument does not exist, and we must consider the effects of real imaging systems in order to ensure that our algorithms will work on real images. Our work began with implementing previously developed techniques that compensate for nonlinear intensity response.

We have also developed methods to compensate for color-filter brightness effects and the limited response range of color video cameras. Color-filter brightness effects derive from the fact that certain filters are optically denser than others. Our new method for ''aperture balancing'' compensates for this phenomenon while preserving the best signal-to-noise ratio the camera can produce. The limited response range of video cameras means that very bright points such as highlights will saturate the camera, producing maximally white data points instead of true image colors. Our reflection analysis method can detect this effect and compensate for it by estimating the true colors, even in areas that have saturated the camera in one or two of the three color bands.

Integrating segmentation and reflection analysis

Our next challenge was to eliminate the heuristic process that must segment an image into individual objects before we can analyze reflections. We addressed this problem by developing a new segmentation method that incorporates the Dichromatic Reflection Model into the earliest stages of visual analysis.

Many computer vision algorithms for image understanding begin by breaking up the scene into regions that correspond to a single object, and then attempting to analyze each of these regions. The problem is that the presence of highlights will fool segmentation algorithms into thinking highlight regions are separate objects. Any analysis program that proceeds from there will have difficulties since the regions will not correctly correspond to individual objects. A preferable method is to analyze the highlights first so they can be correctly recognized as belonging to their surrounding objects.

We showed that according to the model, each pixel neighborhood in the image can be classified into one of several categories such as ''matte'' or ''mixed matte and highlight'' based on the statistics of the colors within the neighborhood. We then developed a variation of region-growing to utilize these classifications in the segmentation process.

As each region is formed, a corresponding hypothesis is generated that describes the way in which the color varies within that region. This description consists of the dimension and direction (principal axes) of the color variation. The criterion for region-growing is to find all the adjacent pixels that conform to a single hypothesis. The result is a kind of region-growing algorithm that differs from the traditional approaches in an important way: instead of grouping pixels by conformance to a single color, the pixels are grouped by conformance with a single color description. Thus all pixels that have the same color dimension and principal axes are considered to belong to the same group. Since the Dichromatic Reflection Model says that plastics will exhibit two colors, the highlight color and the underlying object color, all the pixel neighborhoods corresponding to a given plastic object will have variation along two dimensions and in the same directions. Thus, our algorithm works better than traditional methods, since it successfully recognizes that highlights and shadow regions are parts of an object rather than being separate objects. This is the most successful approach ever proposed for this classical problem in segmentation [Klinker et al. 88a]. The color separation into the two reflection components (highlight and body color) that was done on hand-segmented images in 1987, is now an additional byproduct of the new segmentation method from 1988.

Calculating color-constancy

We also studied how illumination color affects the color measured by a camera. This effect underlies several important problems in low-level vision including color constancy and the estimation of illumination and object colors. We have developed a new mathematical formulation to allow fast and accurate calculation of color constancy and the illumination color. This method uses a system of linear equations based upon constraints generated by the camera measurements of a test target. This mathematical formulation can be used when the image data are noisy; we can use error estimation to determine how to reliably calculate object and illumination colors.

Our method calculates color constancy and illumination color by explicit reference to the full spectrum of color rather than only the red/green/blue (RGB) components. This allows us to model effects such as color metamerism that cannot be modeled in the lower dimensional RGB space. However, in order to do computer processing, the spectral functions need to be represented in a finite manner. We have found that representing the spectral functions as a weighted sum of Legendre polynomials (suggested by Binford and Healey) works much better than sampling the function at regular intervals. This is because we solve the linear constraints using least squares estimation which suffers when variables have a nearly linear relationship. Since most spectral functions are fairly smooth, a given sample is close to equaling a linear combination of nearby samples. This causes trouble for least squares estimation which assumes that variables are independent.

Our method of color constancy allows us to recover a very precise representation of illumination color by use of a test chart with several colors on it. Each observation of a known color gives constraints for the possible spectral make-up of the unknown illumination. Solving for these constraints gives us an estimate of the illumination. In the past, a white card has sometimes been used to give an estimate of the color of the illumination. In a typical camera with three color primaries, this only yields three constraints upon the illumination. A test chart with *n* color patches creates 3*n* constraints in a three-primary camera system. Furthermore, the test chart is a reliable referent, since we know it is in the scene and we know the colors on it. This makes the algorithms more reliable than some previous algorithms which made ad hoc assumptions about images which are not always true (e.g. the brightest point in the scene corresponds to a white object).

Handling image noise

We then applied our algorithm to simulated images which included a model of camera noise. Once the incident illumination has been estimated, the color appearance of any object with known reflectance can be very reliably predicted. In 1989 we showed that our algorithm may be adapted to a neural network model, so that the network solves the constraints. We also showed that our method may be extended to the more general vision problem where the spectral reflectance function of interesting objects is not known in advance. Just as images of several known colors under an unknown illumination give constraints upon the illumination, images of an unknown object under several known (or estimated using our technique) illuminants give constraints upon the object's spectral function, allowing us to solve for these constraints and estimate the object's reflectance function.

One limitation of both our color constancy algorithm and the Dichromatic Reflection algorithm is that they assume that there is only global illumination. This is a common simplifying assumption based upon the model that there is a single light source and that it reflects light onto objects which in turn reflect that light into the camera. In this simple model there is no interaction between objects in the scene. However, in reality, light from one object may reflect onto another, altering the second object's appearance. Often these interreflections will cause a dramatic color change over a very small area of the object.

Exploiting interreflection

We developed a quantitative model of color in scenes with interreflection among multiple objects. We found that our Dichromatic Reflection Model extends naturally to model interreflection. The result is a model of interreflection between two objects in color space. We simulated this type of interreflection between two idealized cylinders and showed the causes of sudden changes in hues observed in interreflection in real objects. In the second half of 1989 we obtained actual images of interreflection between objects and verified that our model is at least *qualitatively* correct — that it captures many of the features that appear in actual images.

Rather than than treating interreflection as a source of noise in the images, we plan to use it as an additional source of information about the scene. We examined how interreflection can give important clues about the surface properties of objects in the scene. The properties of note are roughness, shininess (percentage of light reflected at the surface), and metallic/nonmetallic material type. The image of an object under a single light source may lack information sufficient for determining such surface properties. Since interreflected light follows the same laws of physics as reflected light, other scene objects that are reflecting light onto the primary object can provide additional information about these surface properties.

2.1.2. Understanding texture

To better understand imagery texture and its role in surface and object properties, we are studying the perception of regular texture repetitions. The central issue is a chicken/egg problem; the texture element is difficult to define until the repetition has been detected, but repetition cannot be found until the texture element is defined. To break this cycle, past research has made strict assumptions about either the texture element definition or the repetition pattern. The resulting programs work only in limited kinds of image texture that happen to comply with the assumptions. We are seeking ways to use much more general models: entertaining several hypotheses at one time and using overall relationships among image features.

We have made substantial progress on this approach through combining two developments. The first is the ‘‘dominant feature assumption,’’ which states that a repetitive texture pattern should contain some particular feature within the texture element that is more visually prominent than all the others. By looking for repetitions of just this feature, it is possible to screen away all the other data and arrive at a computationally tractable algorithm for texture analysis. The dominant feature assumption does limit our systems, but it is still quite general. In particular, textures that people perceive easily seem to correspond generally with textures that obey the dominant feature assumption.

The second development is a theory of repetition description that shows how to characterize any two-dimensional repetition using two selected image vectors. The theory is so compact that it has led to a simple algorithm that is provably correct in detecting the vectors that describe any regular two-dimensional repetition. This algorithm uses a new definition of image connectedness based on the six-connected neighborhood graph, a graph that connects the nearest neighbor within each 60-degree sector around the feature points in the image. The graph has simple and important geometrical properties that facilitate its use in analyzing repetitive texture patterns. Furthermore, rather than imposing a single-grid structure on the texture in an image region, our method allows the structure to vary systematically across the region. This generality allows us to apply our method to the deformations (which often vary systematically) that arise in real-world image textures, such as patterns on fabric or perspective texture gradients (foreshortening) on tall buildings. We have successfully applied this method to several images of textured objects.

2.2. Extracting Shape and Reflectance

We have been working towards enhancing our understanding of surface reflection. Most machine vision problems involve analyzing images that result from reflected light. The apparent brightness of a surface point depends on its reflectance characteristics, that is, its ability to reflect incident light in the direction of the sensor. Therefore, image intensity interpretation requires a sound understanding of the various mechanisms involved in the reflection process.

There are two approaches to the study of reflection: physical optics, which uses the wave theory of light to generate accurate models, and geometric optics, which provides a fast method of approximating physical models. In physical optics, the wave theory of light is used to accurately model reflection and diffraction. Geometric optics approximates light as traveling in straight lines, giving much simpler models of optics. While geometric models may be construed as mere approximations to physical models, they possess simpler mathematical forms that often render them more usable than physical models. However, geometric models are applicable only when the wavelength of incident light is small compared to the dimensions of the surface imperfections. Therefore, it is incorrect to use these models to interpret or to predict reflections from smooth surfaces; only physical models are capable of describing the underlying reflection mechanisms. To benefit from the advantages of both approaches, we proposed a unified reflectance model composed of the simple elements from each approach. We then reduced this model to a hybrid model that linearly combines Lambertian (diffuse) and specular (‘‘mirrorlike’’) reflectance components.

Using the hybrid model, we developed a shape and reflectance extraction technique that does not rely on a predetermined reflectance map as do previous shape-from-intensity methods. This new technique is called photometric sampling and consists of eight lights arranged in a circle around an object. These light sources are placed several inches away from the object in the center and are known as *extended light sources*. The extended light sources ensure the detection of both Lambertian and specular components. The image intensities recorded at each surface point are used to compute local estimates of surface orientation and reflectance parameters. The method can therefore adapt to variations in hybrid reflectance from one surface point to another, including the extreme cases of purely Lambertian and purely specular behaviors.

2.3. Visual Depth and Camera Motion

To navigate and act in a poorly known environment, a robot must see. It needs to build a map of its surroundings, in order to know where it is and to plan where to go next. While moving, the robot must be able to verify that the commands issued to its motors produce the desired trajectory with respect to landmarks in the world. In order to manipulate objects carefully, the robot must perceive their shape, usually to a high degree of accuracy.

Although active sensors have been used with increasing levels of performance for these types of tasks, passive, camera-based vision remains attractive in a wide variety of situations. Our research goal has been to use input from mobile cameras to estimate both environment geometry and camera motion. Abstractly, this process amounts to using data from a sequence monocular or binocular images to map the environment and track the camera's trajectory within the map and over time. When it is referred to a sensor-centered frame of reference, this map is usually called a depth map.

This problem has challenged researchers in robotics for a long time. The mathematics of how shape in the world combines with the motion of the camera to produce the flow of patterns in the image is fairly well understood. In spite of this, the inverse process, from images to motion and shape, has been known to be among the hardest in computer vision.

There are two main reasons for this. One is that both shape and motion are very sensitive to noise in the images. The more distant the objects are in the field of view, the more serious this problem becomes. The other reason is that the combination of shape and motion into image sequences is an inherently nonlinear transformation. Shape and motion are tightly coupled to each other, and errors in one quantity propagate to errors in the other, creating instability and lack of convergence to the true values. These two issues, noise sensitivity and coupling of motion and shape, were the two crucial problems we faced.

Our work has shown that the multiframe estimation of depth and motion based on stochastic models is a viable and effective method for obtaining high quality depth maps and a good estimate of the robot's motion. Furthermore, the problem of initializing the estimates is adequately solved by our multistage strategy, which is both practical and accurate for navigation purposes.

2.3.1. Reducing noise sensitivity

Noise sensitivity can be lowered by using more image frames, and having the camera move more between frames. Multiple images are the obvious line of attack when noise is a problem: Statistical estimation has proven a very effective tool, and we chose to exploit its power for the interpretation of visual motion.

The advantage of a wider interframe camera motion, on the other hand, is related to the use of stereo triangulation as the essential means for the recovery of depth. The wider the triangulation baseline, that is, the camera motion between frames, the lower the sensitivity of depth to image noise. Unfortunately, a wider baseline makes it more difficult to identify corresponding points in different images, since weaker geometric constraints can be assumed to hold during the search for correspondences. Furthermore, images taken from distant viewpoints look different from each other and are thus harder to relate.

Thus, there is a dilemma between a wider baseline for good noise rejection and a smaller baseline for an easier correspondence problem. We chose to solve this dilemma by using closely spaced image frames to establish correspondences and obtain initial depth estimates. We then use this noise-corrupted knowledge of depth to restrict the search for correspondences in a wide-baseline stereo system, thus achieving good depth accuracy.

2.3.2. Integrating shape and motion

We approached the other key issue, the coupling of shape and motion, by proposing a solution in stages, so that depth and motion could be effectively be estimated at separate times.

As a concrete basis for the introduction of these ideas, we now sketch the stages in our solution. Initially, the robot does not move. Instead, one camera is translated in a carefully controlled fashion by a positioning platform. As the camera moves, images are taken at closely spaced intervals. Although each pair of successive images, having a small baseline, yields poor depth estimates, the latter can be combined probabilistically into better and better depth maps . Once an approximate depth map is obtained, a calibrated, two-camera stereo system refines the map.

Then, the robot starts moving. Uncertain knowledge of motion from mechanical measurements can be used to propagate the depth map, that is, to change it to reflect the robot's motion. The new depth map can then be used to initialize a new binocular stereo measurement, without using the translation stage, and without having to stop the robot. The stereo triangulation at this stage is relatively easy to perform, because the correspondence problem is simplified by prior depth information from propagation, but leads to accurate measurements, because it is characterized by a wide baseline.

In summary, as the robot navigates, a depth map can be propagated and refined. Given two successive depth maps, a three-dimensional registration algorithm then computes the transformation between the maps, thus refining the coarse motion measurements into an estimate that improves over time.

2.3.3. Developing a stochastic model

To design and analyze this system and to combine the ideas of multiframe estimation and decoupled recovery of depth and motion, we developed a paradigm in which scene depth, camera motion, and image sequences and pairs could be modeled in probabilistic terms. Most previous work in the literature has ignored noise and, more generally, measurement uncertainty, leading to brittle methods that tend to fail when confronted with the noise that arises in real data.

The theory of stochastic dynamic systems proved to be the right formulation for the problem of depth and motion computation. Image noise and uncertainties in the mechanically measured camera motion are modeled as random variables. Their effects on the depth map and on the robot location estimates are analyzed with the techniques of error propagation.

Within this framework, depth and motion estimation can be performed in an incremental fashion by using Kalman filtering techniques. With this approach, the running estimates of both depth and camera motion are updated with each new image, or image pair, that is obtained. In the initial, controlled-motion stage of depth estimation, image displacements are measured by correlating intensity windows in adjacent frames. The depth values obtained by triangulation between frames are then weighted against the current estimates, obtained from previous frames, based on the associated uncertainties. As a result, new depth values and reduced uncertainties are obtained at each step in the camera motion.

Similarly, for the propagation of depth maps and the refinement of motion estimates during robot motion, we use an iconic representation of uncertainty that stores the depth estimate and variance at each pixel. Camera motion, and the resulting changes in the depth map, are described as dynamic systems, whose variations over time are themselves corrupted by noise. Kalman filters then operate in an interleaved manner to incorporate new depth measurements, from stereo triangulation, and motion measurements, from mechanical sensors.

In previous work in the field, the analysis of image sequences could not rely on the appropriate probabilistic representations and algorithms for the combination and refinement of uncertain knowledge. As a consequence, visual motion interpretation was usually performed in a ‘‘batch mode’’ at the end of the motion, thus severely limiting its applicability in realistic applications. With our approach, on the other hand, we obtain results as soon as possible, and refine their quality over time. We compared our depth estimation method both mathematically and experimentally to previous feature-based algorithms. Our results show that lateral camera motion is in fact effective, and that our incremental method can produce results just as good as the best previously known ‘‘batch’’ technique.

Also, the use of an iconic representation of depth produces a denser estimate of depth that is available from previous edge-based techniques, while competing with the convergence rate and accuracy of the latter. Experiments with images of a flat poster have confirmed this analysis and given quantitative measures of the performance of both types of algorithm. Experiments with images of a realistic outdoor-scene model have shown that our algorithm performs well on images with large variations in depth and that even occluding boundaries can be extracted from the resulting depth maps [Matthies et al. 88].

In our current work, we are integrating a model for the correlation between adjacent flow estimates in the images into the Kalman filtering framework. The modeling of correlation, which is presently ignored in our system, is likely to produce more robust and accurate depth estimates, and to improve the convergence speed of the Kalman filter. The central mathematical components of this extension are Bayesian estimation methods applied to random-field models of a range map. We developed the basic mathematical models and operational strategies underlying this extension [Szeliski 88], and are continuing the experimental evaluation of these ideas.

2.4. A Framework for Model-based Computer Vision

Three-dimensional object description and reasoning is critical for many image understanding applications, such as robot navigation and 3-D change detection. A system for 3-D image understanding must include geometric reasoning as a primary component, because geometric relationships among object parts are a rich source of knowledge and constraint for image analysis. Unfortunately, most work in 3-D image understanding has utilized limited solid or surface models and a fixed order of analyzing image features. Such systems perform poorly since they cannot exploit specific properties or relationships within any given image. Our research is aimed at developing a more general framework for representing 3-D models and relationships, so that vision systems can fully exploit specific information contained in each image.

We have been developing a system called 3DFORM, based on the Framekit frame language defined on Common Lisp. The system includes several features that improve on earlier work:

- Extensible models: 3DFORM uses frames to model object parts and geometric relations, which make it easy to extend the system and incorporate new features. The frames are arranged in a class hierarchy, so a new class can be defined by simply specifying differences from existing classes.
- Two-way reasoning: 3DFORM includes explicit modeling of projections from the 3-D scene to the 2-D image and back, permitting a program to reason back-and-forth as needed.
- Optimized computation: Active procedures can be attached to the frames to dynamically compute values as needed, avoiding unnecessary computations.
- Flexible control flow: The order of computation is controlled by accessing objects' attribute values, a strategy that allows the system to perform top-down and bottom-up reasoning as needed.
- Elimination of external ''focus of attention'': There is no need for an external ''focus of attention'' mechanism, which in past systems has sometimes been a complex and problematic item to construct.
- Incremental representation and reasoning: Objects may be specified incompletely (or by placing constraints on them) as opposed to requiring full and complete descriptions.

2.4.1. Combining top-down and bottom-up reasoning

We have been extending the system's capabilities to model more complex parts and relationships such as the relationship of polygon vertices to the lines that form the edges of the polygon. When these relationships are evaluated, the side-effects include creating hypotheses for the missing or incomplete parts of each object. We have also been adding a mechanism for relationships at different levels of the part/whole hierarchy to interact with each other. Then when one feature is added to the interpretation, it can trigger reasoning processes at other levels of the hierarchy. This provides a rich structure for combining top-down and bottom-up reasoning in a single mechanism. We have successfully used our new developments to analyze images in which many of the important features have contrast too low to be analyzed by traditional bottom-up methods.

We applied our initial version of 3DFORM to two task scenarios. The wireframe task begins with partial and noisy sets of line segments derived from an edge operator. The system forms complete 3-D building hypotheses from this data by applying general principles, such as the fact that buildings tend to have vertical walls. 3DFORM is able to create building models even where individual edges may be too low in contrast to be detected reliably, using the context provided by the visible edges plus general 3-D reasoning.

Our other task is geometric interpretation of shadows. We utilize the shadow geometry framework — developed in 1983 as part of our DARPA-sponsored image understanding research — which defines a ‘‘Basic Shadow Problem’’ that occurs repeatedly in complex images. This situation is represented in a 3DFORM frame. The frame is then evaluated wherever the situation occurs in the image. Although this has not yet been integrated into an actual system for aerial photointerpretation, it demonstrates the power of a general 3-D reasoning system for easily assimilating new and relevant theories and applying them to image data.

2.4.2. Merging distinct views

We've also applied our modeling framework to merge 3-D data sets derived from distinct views of a single object [Walker et al. 88]. The system assembles a model for each view, determines whether the models are compatible, and if so, creates a new one satisfying constraints from each source view. In this way, using two partial models, we can create a third compatible with and more complete than either. We combine constraints of multiple models by adding the attributes (including relationship constraints) of one to another, and using attribute slot daemons to determine compatibility. If a particular attribute is constrained to a single value, the value will be automatically computed the next time it is required. We can use the process not only on multiple views of a single object, but also to combine data from multiple sensors, or to match a partial description of a sensed object with a previously entered model while determining the pose of the object.

We then refined portions of the system, especially its ability to maintain equivalence between different representations of the same object. Single subparts can now fill more than one role in modeling complex geometric objects. For example, a horizontal edge at the top of a building is one of the roof edges as well as the top of one wall. Therefore, when 3DFORM hypothesizes a new roof edge, it must also hypothesize a wall with the new edge at its top. We employ the system's matching capability to merge the newly hypothesized wall with one that is already supported by the data. We also use the notion of equivalence when grouping objects into more complex objects. For example, when several walls define a building, the system hypothesizes a roof bounded by the wall tops.

2.5. Acquiring 3-D Recognition Algorithms

Historically, and even today, most successful model-based vision programs are hand-written: Relevant object knowledge is extracted from examples, tailored to the particular environment, and hand-coded into the program. When done properly, the resulting recognizer is effective and efficient, but its development time and demands on machine vision expertise are high.

To simplify the process, we have developed a vision algorithm compiler (VAC) that automatically generates vision programs. Rather than requiring extensive time from highly skilled implementors, a VAC can generate a vision program for a well-defined vision task automatically, given adequate models of objects, sensors, and processing techniques.

Establishing VAC technologies required developing several key components:

- Object models that describe the geometric and photometric properties of a target object
- Sensor models that predict object appearances for a given object/sensor combination
- Strategy generation that uses predicted appearances to prescribe an appropriate recognition methodology
- Program generation that converts a strategy into a executable code.

We have completed designing a VAC for bin-picking tasks and generated several object recognition programs.

2.5.1. Modeling objects

Surveying existing solid modeling systems, we found they suffer two fundamental limitations that preclude their use for our purposes. First, they are designed primarily for the generating display images and therefore do not yield explicit symbolic information about the image structure. Extracting symbolic shape information from the image representation in such systems is difficult, for example. Yet our work requires such information. Second, current systems use closed architectures and hide their internal data structures from the user. This design makes such systems difficult to modify or enhance for sensor modeling.

Thus we embarked on designing a new solid modeler — and appropriate user interface — to support our research. Our system uses the Framekit language and so provides an open architecture and explicit symbolic representations of all the components: sensors, objects, and images. Our Vantage solid modeler offers:

- Explicit representation of 2-D image features (lines, polygons, etc.) in addition to 3-D solid objects. This feature allows us to reason about relationships within the image, crucial for computer vision.
- Implementation as a Common Lisp package allows Vantage to interface with other packages such as low-level vision, graphics display, and robot arm control.

- An open, frame-based architecture makes it easy for vision and robotics researchers to extend data structures as needed for specific research tasks.

2.5.2. Modeling sensors

A recognition program observes an object's 2-D appearance and so requires a corresponding 2-D representation. Since model-based systems frequently use active sensors, their 2-D models must integrate both 3-D object information and sensor limitations. That is, they must be able to represent an object's appearance as the interaction between object model and a model of ‘‘sensor detectability’’ that describes where the sensor can ‘‘see.’’

For our system, we have developed a versatile method of representing sensor detectability. We first define a ‘‘G-source,’’ an abstract sensor component that may be either a light source or a camera, and illumination conditions that specify under what conditions the G-source illuminates surface portions . With this concept, we can represent sensor detectability as a set operation on G-source illumination conditions.

We developed a method to model sensor capabilities and established an interface tool to the Vantage system and implemented sensor modeling in Vantage. Using this framework, we have described stereo vision, photometric stereo, a lightstripe rangefinder, and synthetic aperture radar (SAR) [Ikeuchi and Kanade 88a].

2.5.3. Generating a recognition strategy

With an ability to model a given object and sensor, we next turned to the problem of how to generate an effective recognition strategy. We chose to focus on object localization in bin-picking. There we need analyze only the topmost, usually nonoccluded, object in a jumble of identical parts. We decompose the task into two distinct parts:

- Aspect classification(AC)—An image of an object is classified into one of a small number of topologically distinct appearance groups called aspects. Each aspect represents a collection of viewpoints within which the object ‘‘looks roughly the same’’ according to a mathematical criterion.
- Linear shape change(LC)—With the rough estimate of position and orientation from aspect classification, we can initialize a more accurate procedure that matches model features to observed image features.

Within this framework, a workable VAC technology requires solving three basic problems:

- Aspect generation — How to extract aspects from given object and sensor models?
- Aspect Classification — What kinds of features to use for aspect classification?
- Linear change determination — What kinds of features to use for determining the object's precise attitude and position?

Generating aspects

An ‘‘aspect’’ is a class of objects having topologically equivalent visible structure. We have modified this definition somewhat to describe the class of object appearances having faces visible to a specific sensor. Using this definition, we developed a method that, for given object and sensor models, systematically groups object appearances into like ‘‘aspects’’ [Ikeuchi and Kanade 88b]. Our exhaustive method generates various appearances of an object from uniformly sampled viewer directions under a given sensor model. It then classifies the generated appearances by examining combinations of visible faces. We have implemented this method atop the Vantage geometric/sensor modeler modeler. From a given object and sensor model represented in Vantage, we can generate aspects automatically [Ikeuchi and Kanade 88c].

Classifying aspects

We have developed two different methods to generate an aspect classification strategy. Both methods provide an *interpretation tree* representing what features to examine, and in what order, during aspect classification. Each node represents a classification decision. It stores a necessary feature type and threshold value for the classification. Each leaf node corresponds to one particular aspect.

The first method recursively subdivides possible aspects by available features. Generation of the strategy begins creating a group of all possible aspects. The process examines whether one feature can divide a group of aspects into subgroups. If so, the process registers the features and their threshold values. This operation is applied recursively to a group of aspects along a fixed set of features. The process stops when all groups consist of single aspect or the available features are exhausted. While this method can generate an aspect classification strategy, it offers but no guarantee that the obtained strategy is optimal.

To address the optimality issue, we then developed another method that constructs a minimum cost strategy. Each operation that can be performed during aspect classification was analyzed to determine its computational cost. This enabled the organizing all possible aspect classification strategies as a tree, in which each arc represents an operation and is labeled with the corresponding cost. During compilation, the tree is dynamically constructed and searched using a branch-and-bound method to find the sequence of operations that can perform aspect classification of a given object for the minimum computational cost.

Determining linear change

We have implemented a LC generation module that repeats the following process at each aspect (at each leaf node of an interpretation tree). First, the module examines visible faces at each aspect and determines the most reliable among them. Using the characteristics of the face, it determines a strategy to set up a local coordinate on the face; then, it generates procedures to execute the strategy as well to transform between the local coordinate system and the body coordinate system of the object. Third, it produces a procedure to match visible edges to visible model edges. Finally, it generates a procedure to determine the position and orientation precisely, by iteratively solving the edge-matching equation, given edge correspondences by the previous procedure. These strategies are stored at nodes along a branch of the interpretation corresponding to each aspect.

2.5.4. Program conversion

Strategy generation only provides a recognition strategy. To have a runnable program, we need a method to convert a strategy into executable code. For this, we prepared an program library and a conversion program. The program library is a collection of prototypical ''object-oriented'' objects that can be used to perform feature extractions and threshold operations for aspect classifications and linear change determinations. The conversion program analyzes each node of an interpretation tree, instantiates appropriate objects from the program library, and completes a recognition program.

2.6. Fast Rangefinding

Integrating both sensing and processing on one circuit substrate holds great promise for developing sophisticated acquisition systems. Parallel computing at the point of sensing allows the tailoring of raw data to meet the needs of higher level system requirements. The ability to intelligently acquire data also means that new sensing methodologies can be developed.

The current focus of this research into intelligent sensors involves implementing a high-performance lightstripe range sensor integrated circuit. Due to advances in VLSI technology, we can build smart sensors by integrating sensing and processing. Our experience with this application has shown it to be an ideal testbed for demonstrating the power of combining sensing and processing on monolithic silicon. Our design uses a specialized VLSI sensor that gathers range data as a moving stripe continually sweeps a scene.

We are focusing on a critical component restriction for many robotic applications: rangefinders that measure the three-dimensional profile of an object or scene. The major drawback of conventional lightstripe devices is their slow speed. A faster design, being developed at CMU, is based on the lightstripe rangefinder but extended to record all necessary information. Our design is implemented with VLSI techniques, parallelism, and ''smart'' photosensors. These devices integrate a photoreceptive cell with computing circuitry and associated memory to yield a photosensor that both detects light and processes the signal. Our rangefinding algorithm is practical only because we use such devices. Analog processing techniques have enabled us to achieve a dense arrays of cells that perform the rangefinding function. Instead of using a camera as a sensor, we are now fabricating a VLSI range sensor that contains numerous elements (6×10) combining photosensing, signal conditioning, and signal processing on a single CMOS chip. One photosensor cell needs to be comprised of the following chip components: photoreceptor, analog circuitry, range memory, and signal processing. We have confirmed that theses elements can perform all the necessary functions: sensing the intensity, detecting the stripe, and storing the time of detection.

The VLSI rangefinder uses triangulation to map detectable data points to the 2-D image frame. However, instead of projecting a stripe, processing the information, and then stepping the stripe, the light source emits a plane of light that scans the scene once, left to right. The sensor cell processes incoming data rapidly enough to make stepping action unnecessary. Each cell in the M×M array is oriented so that it can detect those data points that would lie on a predetermined light plane. Additionally, the individual cells record the time (which is why the range memory is necessary) at which the cell's sensing element detected the points lying in some predetermined light plane L. With a lightstripe rangefinder, light plane L can be determined from the angle θ. However, with the VLSI smart photosensor based rangefinder, the light is sweeping across the image at a constant rate rather than being stepped. Given the light plane's (constant) angular velocity, the timestamp gives the photosensor information necessary for correlating detected light points with a light plane.

Work on characterizing the sensitivity, accuracy, and repeatability of range data generated by this 30×30 element range sensor is in progress. Our research to date has produced a lightstripe rangefinding system capable of generating from 100 to 1000 range frames per second — up to two orders of magnitude faster than conventional lightstripe rangefinding methods. The design is based on a specialized VLSI sensor we have designed and fabricated that gathers range data as a moving stripe continually sweeps the scene. One of the most distinguishing features of this approach is that it is not just parallel implementation of known algorithms by VLSI technology to achieve increased speed, such as VLSI chips for convolution. Rather, it demonstrates that the integration principles of information acquisition (in our case, range imaging) results in a qualitative improvement in performance.

2.7. Bibliography

[Chang et al. 88] Chang, H., K. Ikeuchi, and T. Kanade.
Model-based vision system by object-oriented programming.
In *Proceedings of the International Symposium and Exposition on Robots*. November, 1988.

[Choi et al. 90] Choi, T., H. Delingette, M. DeLuise, Y. Hsin, N. Hebert, and K. Ikeuchi.
A Perception and Manipulation System for Collecting Rock Samples.
June, 1990
Presented at Space Operations, Applications and Research Symposium.

[Gremban et al. 88]
Gremban, K., C. Thorpe, and T. Kanade.
Geometric camera calibration using systems of linear equations.
In *Proceedings of the Image Understanding Workshop* . DARPA, April, 1988.

[Hamey 88] Hamey, G.C.
Computer perception of repetitive textures.
PhD thesis, Computer Science Department, Carnegie Mellon University, February, 1988.

[Hamey and Kanade 89]
Hamey, L.G.C. and T. Kanade.
Computer analysis of regular repetitive textures.
In *Proceedings of the DARPA Image Understanding Workshop*. DARPA, May, 1989.

[Hamey et al. 89] Hamey, L., J. Webb, and I. C. Wu.
An Architecture Independent Programming Language for Low-Level Vision.
Computer Vision, Graphics and Image Processing 48(2):246-264, 1989.

[Hong et al. 90] Hong, K.S., K. Ikeuchi, and K.D. Gremban.
Minimum cost aspect classification: a module of a vision algorithm compiler.
Technical Report CMU-CS-90-124, School of Computer Science, Carnegie Mellon University, April, 1990.

[Ikeuchi 87] Ikeuchi, K.
Determining a depth map using a dual photometric stereo.
International Journal of Robotics Research 6(1):15-31, 1987.

[Ikeuchi and Hong 88]
Ikeuchi, K., and K. Hong.
Determining linear shape change: Toward automatic generation of object recognition programs.
Technical Report CMU-CS-88-188, Computer Science Department, Carnegie Mellon University, December, 1988.

[Ikeuchi and Kanade 87]
Ikeuchi, K. and T. Kanade.
Modeling sensor detectability and reliability for model-based vision.
In *Proceedings of the Workshop on Computer Vision*. IEEE, November, 1987.
Also appeared in Proceedings of the Fourth International Symposium of Robotics Research, August 1987, and available as technical report CMU-CS-87-144.

[Ikeuchi and Kanade 88a]
Ikeuchi, K. and T. Kanade.
Modeling sensors and applying sensor model to automatic generation of object recognition program.
In *Proceedings of the Image Understanding Workshop*. DARPA, April, 1988.

[Ikeuchi and Kanade 88b]
Ikeuchi, K., and T. Kanade.
Applying sensor models to automatic generation of object recognition programs.
In *Proceedings of the 2nd International Conference on Computer Vision*. IEEE, December, 1988.

[Ikeuchi and Kanade 88c]
Ikeuchi, K., and T. Kanade.
Automatic generation of object recognition programs.
Proceedings of the IEEE 76(8):1016-1035, 1988.
Also available as technical report CMU-CS-88-138.

[Ikeuchi and Kanade 89]
Ikeuchi, K., and T. Kanade.
Modeling Sensors: Toward Automatic Generation of Object Recognition Program.
Computer Vision, Graphics, and Image Processing 48:50-79, 1989.

[Ikeuchi and Robert 89]
Ikeuchi, K., and J. Robert.
Modeling sensor detectability with VANTAGE geometric/sensor modeler.
Technical Report CMU-CS-89-120, School of Computer Science, Carnegie Mellon University, February, 1989.

[Ikeuchi and Sato 90]
Ikeuchi, K. and K. Sato.
Determining reflectance parameters using range and brightness images.
Technical Report CMU-CS-90-106, School of Computer Science, Carnegie Mellon University, February, 1990.

[Kanade 88] Kanade, T.
CMU Image Understanding Program.
In *Proceedings of the Image Understanding Workshop* . DARPA, April, 1988.

[Kanade and Okutomi 90]
Kanade, T. and M. Okutomi.
A stereo matching algorithm with an adaptive window: theory and experiment.
Technical Report CMU-CS-90-120, School of Computer Science, Carnegie Mellon University, April, 1990.

[Kanade and Shafer 89]
Kanade, T. and S. Shafer.
Image understanding research at Carnegie Mellon.
In *Proceedings of the DARPA Image Understanding Workshop*. DARPA, May, 1989.

[Kanade et al. 87] Kanade, T., C. Thorpe, S. Shafer, and M. Hebert.
Carnegie Mellon Navlab vision system.
Intelligent Autonomous Systems.
North Holland, 1987.

[Kanade et al. 89a]
Kanade, T., A. Gruss, and L.R. Carley.
A VLSI sensor based rangefinding system.
In *Proceedings of the 5th ISRR*. August, 1989.
Robotics Research, H. Miura and S. Arimoto, (eds.). MIT Press.

[Kanade et al. 89b]
Kanade, T., P. Balakumar, J.C. Robert, R. Hoffman, and K. Ikeuchi.
VANTAGE: a frame-based geometric modeler with explicit symbolic representations of 3-D and 2-D information.
In *Proceedings of the 19th ISIR*. November, 1989.

[Klinker et al. 88a] Klinker, G., S. Shafer, and T. Kanade.
Image segmentation and reflection analysis through color.
In *Proceedings of the Image Understanding Workshop* . DARPA, April, 1988.

[Klinker et al. 88b] Klinker, G., S. Shafer, and T. Kanade.
Color image analysis with an intrinsic reflection model.
In *Proceedings of the International Conference on Computer Vision*. IEEE, December, 1988.

[Klinker et al. 88c] Klinker, G., S. Shafer, and T. Kanade.
The measurement of highlights in color images.
International Journal of Computer Vision 2(1):7-32, June, 1988.

[Klinker et al. 90] Klinker, J., S. A. Shafer, and T. Kanade.
A Physical Approach to Color Image Understanding.
International Journal of Computer Vision 4:7-38, 1990.

[Krumm and Shafer 89]
Krumm, J. and S.A. Shafer.
A sampled-grating model of moire patterns from digital imaging.
Technical Report CMU-RI-TR-89-19, The Robotics Institute, Carnegie Mellon University, July, 1989.

[Krumm and Shafer 90]
Krumm, J., and S. A. Shafer.
Local Spatial Frequency Analysis for Computer Vision.
Technical Report CMU-RI-TR-90-11, Carnegie-Mellon University, 1990.

[Matthies and Kanade 87]
Matthies, L., and T. Kanade.
The Cycle of Uncertainty and Constraint in Robot Perception.
Robotics Research: The Fourth International Symposium.
In Bolles and Roth,
MIT Press, 1987.

[Matthies and Kanade 89]
Matthies, L., and T. Kanade.
Kalman filter-based algorithms for estimating depth from image sequences.
International Journal of Computer Vision 3:209-236, 1989.

[Matthies and Okutomi 89]
Matthies, L. and M. Okutomi.
A Bayesian foundation for active stereo vision.
In *Proceedings of SPIE Conference Sensor Fusion II: Human and Machine Strategies*. Society of Photo-optical Instrumentation Engineers, November, 1989.

[Matthies et al. 88] Matthies, L., R. Szeliski, and T. Kanade.
Incremental estimation of dense depth maps from image sequences.
In *Proceedings of the Conference on Computer Vision and Pattern Recognition 1988* . IEEE, June, 1988.

[McKeown 87a] McKeown, D.M. Jr.
Automated compilation of spatial knowledge from image analysis.
Image Understanding Systems and Applications 758:144-164, 1987.

[McKeown 87b] McKeown, D.M. Jr.
The role of Artificial Intelligence in the integration of remotely sensed data with geographic information systems.
IEEE Transactions on Geoscience and Remote Sensing GE-25(3):330-348, 1987.

[Milenkovic 88] Milenkovic, V.
Verifiable Implementations of Geometric Algorithms using Finite Precision Arithmetic.
PhD thesis, Computer Science Department, Carnegie Mellon University, July, 1988.

[Nayar and Ikeuchi 89]
Nayar, S., and K. Ikeuchi.
Photometric Sampling: A Method for Determining Shape and Reflectance of Surfaces.
Machine Vision for Inspection and Measurement.
Academic Press, 1989.

[Nayar et al. 89a] Nayar, S.K., K. Ikeuchi, and T. Kanade.
Surface reflection: physical and geometrical perspectives.
Technical Report CMU-RI-TR-89-7, The Robotics Institute, Carnegie Mellon University, March, 1989.

[Nayar et al. 89b] Nayar. S.K., K. Ikeuchi, and T. Kanade.
Extracting shape and reflectance of hybrid surfaces by photometric sampling.
In *Proceedings of the DARPA Image Understanding Workshop*. DARPA, May, 1989.

[Nayar et al. 89c] Nayar, S., K. Ikeuchi, and T. Kanade.
Determining Shape and Reflectance of Lambertian, Specular and Hybrid Surfaces Using Extended Surfaces.
In *International Workshop on Machine Intelligence and Vision*. April, 1989.

[Nayar et al. 89d] Nayar, S., K. Ikeuchi, and T. Kanade.
Shape and Reflectance from an Image Sequence Generated Using Extended Sources.
In *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*. IEEE, May, 1989.

[Nayar et al. 90] Nayar, S., K. Ikeuchi, and T. Kanade.
Shape from Interreflections.
Technical Report CMU-RI-TR-90-14, Carnegie-Mellon University, 1990.

[Novak and Shafer 90]
Novak, C.L. and S.A. Shafer.
Supervised color constancy using a color chart.
Technical Report CMU-CS-90-140, School of Computer Science, Carnegie Mellon University, June, 1990.

[Novak et al. 90] Novak, C.L., S.A. Shafer, and R.G. Willson.
Obtaining accurate color images for machine vision research.
In *Proceedings of the SPIE Conference Vol. 1250, Perceiving, Measuring, and Using Color*. Society of Photo-optical Instrumentation Engineers, February, 1990.

[Reece and Shafer 90]
Reece, D.A. and S. Shafer.
The impact of domain dynamics on intelligent robot design.
Technical Report CMU-CS-90-130, School of Computer Science, Carnegie Mellon University, May, 1990.

[Shafer 88] Shafer, S.A.
Automation and Calibration for Robot Vision Systems.
Technical Report CMU-CS-88-147, Carnegie-Mellon University, May, 1988.

[Shafer 89] Shafer, S.A.
Geometric camera calibration for machine vision systems.
Manufacturing Engineering 102number=3month=mar:85-88, 1989.

[Shafer et al. 90] Shafer, S.A., T. Kanade, J. Klinker, and C. Novak.
Physics-Based Models for Early Vision by Machine.
In *SPIE Conference on Perceiving, Measuring and Using Color* . February, 1990.

[Szeliski 88] Szeliski, R.S.
Bayesian modeling of uncertainty in low-level vision.
PhD thesis, Computer Science Department, Carnegie Mellon University, August, 1988.
Also available as technical report CMU-CS-88-169.

[Thorpe et al. 87] Thorpe, C., S. Shafer, and T. Kanade.
Vision and navigation for the Carnegie Mellon Navlab.
In *Proceedings of the American Institute of Aeronautics and Astronautics*. AIAA, 1987.

[Tomasi and Kanade 90]
Tomasi, C. and T. Kanade.
Shape and motion without depth.
Technical Report CMU-CS-90-128, School of Computer Science, Carnegie Mellon University, May, 1990.

[Walker 89] Walker, E.G.L.
Frame-based geometric reasoning for construction and maintenance of 3D world models.
Technical Report CMU-CS-89-177, School of Computer Science, Carnegie Mellon University, August, 1989.

[Walker and Herman 87]
Walker, E. and M. Herman.
Geometric reasoning for constructing 3-D scene descriptions from images.
In *Proceedings of the Workshop on Spatial Reasoning and Multisensor Fusion*. AAAI, October, 1987.

[Walker et al. 88] Walker, E., M. Herman, and T. Kanade.
A framework for representing and reasoning about three-dimensional objects for vision.
AI Magazine 9(2):47-58, 1988.

[Wallace 89] Wallace, R.S.
Finding natural clusters through entropy minimization.
PhD thesis, School of Computer Science, Carnegie Mellon University, June, 1989.
Also available as technical report CMU-CS-89-183.

[Wallace and Kanade 89]
Wallace, R.S. and T. Kanade.
Finding hierarchical clusters by entropy minimization.
In *Proceedings of the DARPA Image Understanding Workshop*. DARPA, May, 1989.

[Wallace et al. 88] Wallace, R., J. Webb, and I-C. Wu.
Machine-independent image processing: performance of Apply on diverse architectures.
In *Proceedings of the Image Understanding Workshop*. DARPA, April, 1988.

3. RESEARCH IN RELIABLE DISTRIBUTED SYSTEMS

Our work in Reliable Distributed Systems aims at simplifying the construction of distributed applications that access shared data, particularly those that require continued operation despite the occurrence of failures. Our strategy was to develop a distributed transaction facility and associated linguistic support. This effort was divided into four major subgoals:

- Develop a machine-independent, high-performance, distributed transaction facility (Camelot) for uni- and multi-processors that provides systems support for coordinating and controlling access to data in a large heterogeneous environment.
- Design and implement a set of appropriate high-level programming language primitives (Avalon).
- Devise formal methods for reasoning about such application programs.
- Demonstrate the utility of our approach via implementing practical applications and algorithms.

3.1. Background

Camelot and Avalon build on the transaction model of distributed computing. A *distributed system* consists of multiple computers (called *nodes*) that communicate through a network. Distributed systems are typically subject to several kinds of failures: nodes may crash, perhaps destroying local disk storage, and communications may fail, via lost messages or network partitions. A widely-accepted technique for preserving consistency in the presence of failures and concurrency is to organize computations as sequential processes called *transactions*. A transaction is a collection of operations that reduce the attention a programmer must pay to concurrency and failures, by providing three properties:

- **Failure atomicity**: If a transaction's work is interrupted by a failure, any partially completed results will be undone. A programmer or user can then attempt the work again by re-issuing the same or a similar transaction.
- **Permanence**: If a transaction completes successfully, the results of its operations will never be lost, except in the event of catastrophe. Systems can be designed to reduce the risk of catastrophe to any desired probability.
- **Serializability**: Transactions are allowed to execute concurrently, but the results will be the same as if the transactions executed serially. Serializability ensures that concurrently executing transactions cannot observe inconsistencies. Programmers are therefore free to cause temporary inconsistencies during the execution of a transaction knowing that their partial modifications will never be visible.

It is also assumed that programmers write transactions so that they will take the database from one consistent state to another. With this consistency assumption and the failure atomicity, permanence, and serializability properties, databases are guaranteed to remain consistent across failures. Frequently, the three transaction properties and the consistency assumption are called the four ACID properties of a transaction: Atomicity, Consistency, Isolation (serializability), and Durability (permanence).

3.2. Camelot

Most computer systems projects have a simple, central goal. However, systems designers must then balance particular (lower-level) functional and performance goals (or specifications) against the constraints that govern a system's development and operation. Our central goal in the Camelot Project was to develop a system embodying innovative techniques for simplifying the development of reliable distributed programs. There was no doubt that Camelot would utilize transaction technology.

Initially, the project needed to specify the precise collection of functions that Camelot would support; Also, performance specifications had to be developed. Fortunately, these jobs were made easier since many intended uses for the Camelot system were known. Early on, the development and target computing environments were constrained to be Mach and **C**.

Because most project members had worked on a similar system, there was a crucial head start in understanding the complexity and performance of different collections of potential functions. It was quickly realized that Camelot would require the development of 50,000 to 100,000 lines of new computer code, so great care had to be exercised to keep the complexity to a minimum.

A major goal of the top-level architecture of Camelot is to reduce the number of times that messages need to be exchanged between Mach tasks. This leads to a decomposition in which commonly performed functions are performed by the Camelot Library without the need to send messages. (In particular, servers may lock and modify objects without sending messages.) It also leads to the merging of the local Log Manager and Disk Manager into one task. Shared memory is not used extensively in Camelot, because the correctness of applications and data servers cannot be assumed. Since Camelot must protect itself and its clients, it must execute in protected address spaces.

Code modularity was the second most important goal, and this led to the division of the Recovery Manager, Transaction Manager, and Disk Manager into separate tasks. The Recovery Manager is not active and hence rarely communicates with any task, so it was an obvious candidate to become a separate task. The Transaction Manager and Disk Manager communicate only once or twice for write transactions. The Transaction Manager and Communication Manager communicate frequently and have to maintain much shared state, so we ultimately determined that the two should be combined into one address space. The Camelot Project decided that the benefits of being able to more easily develop separate tasks outweighed the slight performance benefits that would have accrued from having a single, large task.

3.2.1. Performance Goals

Camelot's design supports secure, authenticated inter-node communication; control of parallelism on a node, including the use of shared-memory multiprocessors; inter-transaction synchronization; and transaction recovery after transaction, server, node, and media failures.

Camelot's performance goals apply to both normal operation and recovery after failures. The performance goals for transactions that eventually commit are as follows:

- **I/O of recoverable data.** Camelot should not add appreciably to the cost of normal I/O, and should permit servers to supply knowledge of data access patterns, thereby permitting even more efficient I/O. Additionally, Camelot must be able to perform parallel I/O to disks, up to the limits of the underlying operating system.
- **Transaction execution.** The per-transaction overhead of top-level transactions should be sufficiently low to make even short transactions feasible. The per-transaction overhead of nested-transactions should be a small constant that does not preclude their wide use.
- **Operation calls.** Camelot should not add appreciable overhead to normal remote procedure calls (RPCs).
- **Synchronization.** The cost of obtaining a free or shared logical lock should be very low: less than a thousand instructions. Access to locks previously held by other transactions may be much higher, particularly if they have been held by transactions within the same transaction family.
- **Operation on multiprocessors.** Camelot should not have bottlenecks that preclude the efficient use of shared memory multiprocessors.
- **Checkpointing.** Camelot should be able to perform checkpoints and the associated flushing of dirty pages efficiently, so as to reduce the number of log records that need to be considered during node recovery.

In all instances, Camelot should provide maximal throughput by permitting overlapped use of I/O devices, processors, and networks.

Camelot's recovery processing performance goals are the following:

- **Transaction abort.** Camelot's processing of aborts should require roughly the same time as the Camelot system time in forward processing. Importantly, nodes should never have to wait for other nodes to recover before finishing an abort.
- **Node failure.** Recovery after node failure should require 10% to 100% of the cost of forward processing. The 100% cost should occur only when forward processing is doing little else but streaming log records at full speed. With checkpoint and hot-page flushing reasonably occurring every 10 minutes or so, node recovery should require between 1 and 10 minutes.
- **Media failure.** Media failure recovery should require only the time to transfer the archival dump file plus at most 10% of the execution time since the most recent archival dump was written. This could be on the order of an hour for heavily-used databases with nightly dumps.

- **Continued forward processing.** Forward processing should continue during transaction abort processing. Forward processing on operational servers should be possible during node or media recovery of other servers.

3.2.2. Implementation

The design and implementation of Camelot were heavily influenced by the design, implementation, and schedule of Mach. The latter affected Camelot since some features of Mach were developed only as Camelot matured.

The feature of both UNIX and Mach that most influenced Camelot is the necessity of implementing separate protected components as separate tasks. The high cost of context switching implied that Camelot had to be designed to minimize the number of times that RPCs are used to cross subsystem boundaries. On Mach, calling one local subsystem from another via RPC is about 100 to 1000 times slower than issuing a local procedure call. Sometimes Camelot's design or implementation combines two logically separate functions into one task to reduce inter-task communication. In some instances, Camelot also uses Mach's shared memory for communication.

Calling one subsystem from another *non-local* subsystem is 1000 to 10000 times slower than issuing a local procedure call. This influenced Camelot to reduce the number of non-local RPCs to the minimum and to use UDP datagrams in the Transaction Manager and the distributed Log Manager.

Control of paging I/O on Mach requires the addition of an external memory manager task. I/O to secondary storage must ultimately be done using the UNIX file system calls, to either the raw or buffered file system.

In addition, the use of Camelot had to be natural for UNIX **C** programmers and compatible with other uses of UNIX, and Camelot's design had to be modular enough from the beginning to accommodate change during implementation.

3.2.3. Results

Camelot is a working, well-integrated system that provides support for distributed transactions. This support is provided by the following four major facilities.

Node Configuration

Camelot supports dynamic allocation and deallocation of both new data servers and the *recoverable storage* in which data servers store long-lived objects. At every node, Camelot maintains a collection of configuration data to support this dynamic activity. This configuration data contains a list of the data servers that should be restarted after a crash, the recoverable storage to which they should be attached, and their recoverable storage allocation limits. These configuration data are stored in recoverable storage and may be updated transactionally by properly authorized users.

Library Support for Data Servers and Applications

The Camelot Library is composed of routines and macros that allow a user to implement data servers and applications. For servers, it provides a common message handling framework and standard processing functions for system messages. Thus, the task of writing a server is reduced to writing procedures for the operations supported by the server.

The Library provides several categories of support routines to facilitate the task of writing these procedures. Transaction control routines provide the ability to initiate and abort top-level and nested transactions. Data manipulation routines permit the creation and modification of static recoverable objects. Locking routines maintain the serializability of transactions. (Lock inheritance among families of subtransactions is handled automatically.) Critical sections control concurrent access to local objects. Macros facilitate RPCs to other servers.

Recoverable Storage

Camelot provides data servers with up to 2^{48} bytes of recoverable storage. Camelot also provides data servers with logging servers for recording modifications to objects. These services allow modifications of recoverable storage to be undone or redone after failures so that failure atomicity and permanence guarantees can be met.

Values in a transaction are logged in one of two forms: either only new values in a transaction are logged, or both old and new values are logged. In comparison with old-value/new-value logging, new-value logging requires less log space, but increases paging for long-running transactions. This is because pages cannot be written back to their home location until a transaction commits. Camelot assumes that the invoker of a top-level transaction knows the approximate length of the transaction and will accordingly specify the type of logging.

Camelot also provides utilities to save and restore archival dumps of recoverable storage. Archival dumps limit the amount of log space that is needed to recover from media failures.

Transaction Management

Camelot provides facilities for beginning, committing, and aborting new top-level and nested transactions.

- When a top-level transaction is begun, the transaction can be permitted to invoke operations on any number of servers, or it can be restricted to the server that initiates the transaction. In the latter case, the transaction is called *server-based* and it has substantially less overhead.
- When a transaction attempts to commit, and *blocking*, *nonblocking*, or *lazy* commit protocol can be specified. Blocking (two-phase) commit guarantees failure atomicity and permanence, but failures may cause data to remain locked until a coordinator is restarted or a network is repaired. Nonblocking commit, though more expensive in the normal case, reduces

the likelihood that a node's data will remain locked until another node or network partition is repaired. Lazy commit is only for transactions that are local to a server, and it does not guarantee permanence of effect until another transaction (on the same node) has later committed with either a blocking or nonblocking commit.

- Both the system and users can initiate aborts. User aborts can be used to abort either the innermost nested transaction or an entire top-level transaction. Abort calls take a status variable as an argument that Camelot will propagate to all sites involved with the transaction.

In addition to these standard transaction management functions, Camelot provides an inquiry facility for determining a transaction's status.

Security and Authentication

Camelot provides an integrated set of tools, collectively called *Strongbox*, that provide end-to-end client/server authentication and encryption. Strongbox provides programmers with strong guarantees as to the privacy and integrity of their data. However, Strongbox does not prevent traffic or denial of service attacks.

Strongbox primitives are very similar to Camelot Library primitives, making them easy to use. Because Strongbox is layered on top of Camelot, programmers are free to choose whether or not they want to use the security facility it provides.

3.2.4. Design and Implementation Weaknesses

Camelot's design goals have proved to be reasonable. However, Camelot would be more useful if it met some additional requirements:

- **An open log.** The Camelot Log Manager permits use only by the Camelot Disk Manager, Recovery Manager, and Transaction Manager. While recoverable storage can obviate the need for other recovery mechanisms, servers may nonetheless wish to implement their own recovery techniques using a private buffer pool and recovery algorithm. Having interfaces to permit them to read from and write to the common log would be valuable to them.
- **Better portability and interoperability.** Many Camelot components (e.g., the Transaction Manager) could have been implemented more portably, increasing the utility of the Camelot code base. Camelot would be more useful if it supported access to transaction services on other platforms.
- **More flexible locking.** A locking mechanism that better supported intention locks would be a useful addition to the Library.

In retrospect, the choice of algorithms for Camelot seems to have been reasonable. Despite the fact that the Camelot team did not understand the complexity of efficient checkpointing, distributed abort, and communication until near the end of the Camelot implementation effort, the resultant algorithms appear to work satisfactorily and to be reasonably close to what is needed.

However, there are a number of implementation details that did not turn out well. For example, the recoverable storage allocator has too much overhead, and a simpler storage allocation algorithm should be used. Insufficient attention was paid to the performance of recovery, so it can be slow under some circumstances. Many Camelot components are overly complicated and should be rewritten to simplify them, and enhance their reliability. For example, the Transaction Manager should be rewritten to be table driven. Overall, the system is not reliable, and an enormous amount of additional work would be required to make it so.

The performance evaluation of Camelot has many gaps in it, despite the substantial amount of work that has been done. For example, when running the ET-1 benchmark, there are occasional transactions that become delayed for a second or so, and the reason is still unclear. Also, there has been little evaluation of the use of Camelot on multiprocessors. Camelot is designed to effectively use multiple CPUs per node, but this aspect of the design has hardly been tested.

Perhaps the greatest weakness of Camelot was the lack of component testing as the system was developed. Due to a lack of formal code review and testing processes, too many bugs were found via the stress testing of the complete system. Because of the complexity of a system like Camelot, this lack of a sufficiently careful development methodology will prevent Camelot from achieving the reliability required for production systems.

3.3. Avalon

Programming reliable distributed systems is inherently more difficult than programming conventional sequential systems because of the complexity introduced by concurrency and failures. Camelot is a very large, well-integrated system for transactions. However, a typical applications programmer is not going to be able to very easily use that entire system. The Avalon project at Carnegie Mellon is intended to help programmers master this complexity by allowing them to implement and reason about programs in terms of high-level constructs meaningful to the application, while still exploiting the efficiency and flexibility of Camelot and Mach. Application writers can build abstract atomic types and control abstractions without worrying about lower-level details such as transaction management and storage stability. Work on specifying and verifying this linguistic support (in the form of language extensions) enhances both our understanding of our extensions' semantics as well as the confidence of the programmers who use them.

The driving idea behind Avalon's programming language design is to identify the right constructs that give the expressibility to the programmer, and hide the complexity of the lower-level details. For instance, a programmer using one of the Avalon extensions might actually have no idea of the details of the recovery algorithm, but will still be able to write an application using the operation called "recover." Since this operation can actually be called under the covers by the runtime system (is the environment in which a low-level, compiled program runs), when a failure occurs the programmer knows that "recovery will happen" somehow, without having to be aware of the underlying intricacies of Camelot and Mach. Avalon aims to provide this kind of access to Camelot's power with extensions to the **C**$^{++}$ and Lisp languages, and with underlying runtime support.

3.3.1. Avalon/C^{++}

A program in Avalon consists of a set of *servers*, each of which encapsulates a set of objects and exports a set of *operations* and a set of *constructors*. A server resides at a single physical node, but each node may be home to multiple servers. An application program may explicitly create a server at a specified node by calling one of its constructors. Rather than sharing data directly, servers communicate by calling one another's operations. An operation call is a remote procedure call with call-by-value transmission of arguments and results. Objects within a server may be *stable* or *volatile*; stable objects survive crashes, while volatile objects do not. Avalon/**C**$^{++}$ includes a variety of primitives for creating transactions in sequence or in parallel, and for aborting and committing transactions. Each transaction is the execution of a sequence of operations; each is identified with a single process.

A design decision in Avalon was to not invent another programming language. It is unlikely that anyone would stop programming in a language with which they are familiar in order to use Camelot. We targeted specific existing programming languages and worked to add extensions, providing linguistic constructs which access Camelot facilities.

Avalon/**C**$^{++}$ adds extensions to **C**$^{++}$ to give the **C**$^{++}$ programmers the right linguistic constructs to exploit the facilities provided by Camelot. Camelot serves as the runtime system of the language.

Avalon/**C**$^{++}$ takes a piece of Avalon code and translates it into **C**, which then executes. Much of that **C** code consists of calls to the Camelot **C** interface; hence, Camelot becomes Avalon's runtime environment. An example of the power of Avalon programming language support is provided by one application which we wrote, consisting of about 350 lines of Avalon/**C**$^{++}$ code. That 350 lines translates into about 10,000 lines of real code in **C**. There is additional benefit from this, in that there are consequently fewer places for a programmer to make a mistake. The 10,000 lines of **C** code are actually much farther removed from the application.

The only other instance of programming language support for a transaction-based system is a project called Argus from MIT. We support a notion of correctness that permits more concurrency than Argus.

For example, Avalon allows concurrent transactions where those transactions might be enqueuing and dequeuing onto a shared queue. Intuitively, it seems that one process should be allowed to remove an object from the end of a queue, at the same time letting another process add an object onto the other end, as long as the queue is not empty. The two processes are working at different ends of the queue, so they shouldn't interfere. Argus does not directly allow that, because it has a simpler notion of correctness. It says that any time a process is writing to a shared object, it locks out all other writers. Enqueuers and dequeuers are both modifying this shared object, so they both are classified as writers. Therefore, Argus disallows concurrent enqueuers and dequeuers. Avalon also supports data structures other than queues, such as trees. These are very easy to write in Avalon code, permitting higher degrees of concurrency.

Another technical difference between Avalon and Argus is that in Argus the application writer has absolutely no control over what follows a commit or abort. The programmer cannot specify "upon an abort, do X." Avalon gives the user explicit control over commit and abort, and provides the user with the capability to fine-tune what happens upon a commit or abort of the transaction.

Avalon also allows the programmer to determine the status of the transaction at runtime. The transaction can have committed, aborted or still be active. We provide a class in Avalon/**C**$^{++}$ called *trans_id* which lets the programmer determine whether a transaction has committed with respect to the querying transaction. Providing this support empowers the programmer with respect to increasing the degree of concurrency.

Another critical improvement over the Argus system derives from the fact that **C**$^{++}$ is an object oriented programming language which supports the notion of inheritance. Avalon is the first language to exploit the use of inheritance for this realm of fault-tolerant distributed transaction-based computing. In particular, the most visible realization of this is the class-hierarchy that we provide in **C**$^{++}$. Each of the classes built into Avalon/**C**$^{++}$ can be considered a **C**$^{++}$-like class. From these three classes, all other objects are derived.

3.3.2. Avalon/Common Lisp

Our work in Avalon/Common Lisp is part of the same effort not to invent a new language, but to target existing languages. Avalon/Common Lisp is an attempt to provide a similar kind of veneer to that provided by Avalon/**C**$^{++}$, but to the Lisp community, to provide them access to Camelot's functionality. Avalon/Common Lisp provides support for remote evaluation. Suppose one has a computation running at a local site, and wants to exploit the resources at some remote site. Instead of doing a remote procedure call and encountering the expense of shipping data back and forth to compute a function locally, the function is shipped to the remote site where the data is located. The results are then shipped to the local site where the computation was initiated. In Lisp, functions are treated as first-class data. This enabled us to implement remote evaluation, whereas in **C** it would have been impossible.

Another technical innovation of Avalon/Common Lisp is support for a more generalized client-server model. In this model, both the client and the server can be split between a local and remote site. This split or generalization of the standard client-server model results in greater efficiency. Part of the client might be defined remotely, and call those remotely defined functions to execute remotely. Part of the server might be defined locally, so that the whole bundle of messages can be eliminated by a local call.

3.3.3. An interface to Camelot's functionality.

The primary motivation for our work in Avalon is to provide programming language support for any very large software system. Camelot is a very large, well- integrated, system for transactions. However, a typical applications programmer is not going to be able to very easily use the entire system because of its inherent complexity—one must take the entire Camelot package, not just one or two useful tools. Avalon abstracts from the morass of Camelot the high-level concepts and encapsulates those high-level concepts in linguistic constructs that an applications programmer can easily use, leaving all low-level operating system intricacies hidden. For instance, Camelot itself runs on top of Mach, and the Avalon programmer need not know anything about Mach to use Avalon.

3.4. Verifying Atomic Data Types

We have formulated proof techniques that allow programmers to verify the correctness of atomic objects in a transaction-based system. Although language and system constructs for implementing atomic objects have received considerable attention in the distributed systems community, the problem of verifying the correctness of programs have received surprisingly little attention. To our knowledge, the Avalon Project is the only language project to address this particular program verification problem. The significant aspect of the developed technique is the extension of Hoare's abstraction function to map a set of abstract operations, not just to a single abstract value.

We have also used the Larch Prover to prove the correctness of a non-trivial implementation of a highly concurrent FIFO queue. The queue derives from class subatomic and we proved it satisfies the hybrid atomicity property, as must be shown of all Avalon objects. The Larch traits, which include axiomatization of much of Avalon's model of computation, were three pages long; the proof transcripts, which includes proofs of helping lemmas, were 168 pages.

3.5. Bibliography

[Barbacci and Wing 87a]
Barbacci, M.R. and J.M. Wing.
Durra: a task-level description language.
In *Proceedings of the International Conference on Parallel Processing*. 1987.
Also available as Technical Report CMU-CS-86-176.

[Barbacci and Wing 87b]
Barbacci, M.R. and J.M. Wing.
Specifying functional and timing behavior for real-time applications.
Lecture Notes in Computer Science, Vol. II, Parallel Languages. Proceedings of PARLE (Parallel Architectures and Languages Europe) 259.
Springer-Verlag Publishers, 1987.
Also available as Technical Report CMU-CS-86-177.

[Barbacci and Wing 90]
Barbacci, M.R. and J.M. Wing.
A Language for Distributed Applications.
In *Proceedings of the ACM Workshop on Formal Methods in Software Development*. March, 1990.

[Barbacci et al. 88a]
Barbacci, M.B., C. B. Weinstock, and J. L. Wing.
Programming at the Processor-Memory- Switch Level.
In *Proceedings of the Tenth International Conference on Software Engineering*. March, 1988.

[Barbacci et al. 88b]
Barbacci, M.R., M. P. Herlihy, and J. L. Wing.
CMU Software Engineering Institute Special Report.
Technical Report CMU-CS-TR-112, Computer Science Department, Carnegie Mellon University, February, 1988.

[Bhandari et al. 87]
Bhandari, I.S., H.A. Simon, and D.P. Siewiorek.
Optimal diagnosis for causal chains.
Technical Report CMU-CS-87-151, Computer Science Department, Carnegie Mellon University, September, 1987.

[Black 90]
Black, D.L.
Scheduling and Resource Management Techniques for Multiprocessors.
PhD thesis, School of Computer Science, Carnegie Mellon University, July, 1990.

[Bloch 89a] Bloch, J.J.
The Camelot library: A **C** language extension for programming a general purpose distributed transaction system.
In *Proceedings of the Ninth International Conference on Distributed Computing Systems*. June, 1989.

[Bloch 89b] Bloch, J.J.
The Camelot library.
Guide to the Camelot distributed transaction facility, including the Avalon language.
In J.L. Eppinger, L.B. Mummert, and A.Z. Spector,
Prentice-Hall, 1989.

[Bloch 89c] Bloch, J.J.
The design of the Camelot library.
Guide to the Camelot distributed transaction facility including the Avalon language.
In Eppinger, J.L., L.B.Mummert, and A.Z. Spector,
Prentice-Hall, 1989.

[Bloch 90] Bloch, J.J.
A practical approach to replication of abstract data objects.
PhD thesis, School of Computer Science, Carnegie Mellon University, May, 1990.
Also available as technical report CMU-CS-90-133.

[Bloch et al. 87] Bloch, J.J., D.S. Daniels, and A. Spector.
A Weighted Voting Algorithm for Replicated Directories.
BDSRepDir 34(4), 1987.

[Burch et al. 90a] Burch, J.R., E.M. Clarke, K.L. McMillan, and D.L.Dill.
Sequential circuit verification using symbolic model checking.
In . ACM/IEEE, June, 1990.

[Burch et al. 90b] Burch, J.R., E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang.
Symbolic model checking: 10^{20} states and beyond.
In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*. June, 1990.

[Clamen et al. 89a]
Clamen, S.M., L.D. Leibengood, S.M. Nettles, and J.M. Wing.
Reliable distributed computing with Avalon/Common Lisp.
Technical Report CMU-CS-89-186, School of Computer Science, Carnegie Mellon University, September, 1989.

[Clamen et al. 89b]
Clamen, S.M., L.D. Liebengood, S.M. Nettles. and J.M. Wing.
An overview of Avalon/CommonLisp.
In *Proceedings of the 3rd Workshop on Large Grained Parallel Programming, Pittsburgh, PA*. October, 1989.

[Clamen et al. 90] Clamen, S.M., L.D. Leibengood, S.M. Nettles, and J.M. Wing.
Reliable Distributed Computing with Avalon/Common Lisp.
In *Proceedings of IEEE Computer Society 1990 International Conference on Computer Languages*. IEEE, March, 1990.
Also available as technical report CMU-CS-89-186.

[Daniels 89a] Daniels, D.S.
Distributed logging for transaction processing.
Technical Report CMU-CS-89-114, School of Computer Science, Carnegie Mellon University, January, 1989.

[Daniels 89b] Daniels, D.
The design of the Camelot distributed log facility.
Guide to the Camelot distributed transaction facility including the Avalon Language.
In Eppinger, J.L., L.B.Mummert, and A.Z. Spector,
Prentice-Hall, 1989.

[Daniels et al. 87] Daniels, D.S., A.Z. Spector, and D.S. Thompson.
Distributed logging for transaction processing.
In *Sigmod '87 Proceedings*. ACM, May, 1987.
Also available as technical report CMU-CS-86-106.

[Detlefs 90] Detlefs, D.L.
Concurrent garbage collection for **C**$^{++}$.
Technical Report CMU-CS-90-119, School of Computer Science, Carnegie Mellon University, May, 1990.
A shorter version of this report will appear in *Topics in Advanced Language Implementation*, P. Lee, ed., MIT Press.

[Detlefs et al. 87] Detlefs, D.L., M.P. Herlihy, K.Y. Keitzke, and J.M. Wing.
Avalon/**C**$^{++}$: C++ extensions for transaction-based programming.
In *Proceedings of the 1986 USENIX Workshop on* **C**$^{++}$. USENIX, November, 1987.

[Detlefs et al. 88] Detlefs, D.L., M.P. Herlihy, and J.M. Wing.
Inheritance of synchronization and recovery properties in Avalon/**C**$^{++}$.
IEEE Computer 21(12), 1988.
Also available as technical report CMU-CS-87-133, and appeared in *Proceedings of the Hawaii International Conference on System Sciences*, August, 1988.

[Duchamp 88] Duchamp, D.
Transaction management.
PhD thesis, School of Computer Science, Carnegie Mellon University, December, 1988.

[Duchamp 89] Duchamp, D.
The design of the Camelot transaction manager.
Guide to the Camelot distributed transaction facility including the Avalon language.
In Eppinger, J.L., L.B.Mummert, and A.Z. Spector,
Prentice-Hall, 1989.

[Eppinger 89] Eppinger, J.L.
The design of the Camelot disk manager.
Guide to the Camelot distributed transaction facility including the Avalon language.
In Eppinger, J.L., L.B.Mummert, and A.Z. Spector,
Prentice-Hall, 1989.

[Eppinger and Menees 89]
Eppinger, J.L., and S.G. Menees.
Recoverable storage management in Camelot.
Guide to the Camelot distributed transaction facility including the Avalon language.
In Eppinger, J.L., L.B.Mummert, and A.Z. Spector,
Prentice-Hall, 1989.

[Eppinger and Michaels 89a]
Eppinger, J.L., and G. Michaels.
Camelot node configuration.
Guide to the Camelot distributed transaction facility including the Avalon language.
In Eppinger, J.L., L.B.Mummert, and A.Z. Spector,
Prentice-Hall, 1989.

[Eppinger and Michaels 89b]
Eppinger, J.L., and G. Michaels.
Camelot node management.
Guide to the Camelot distributed transaction facility including the Avalon language.
In Eppinger, J.L., L.B.Mummert, and A.Z. Spector,
Prentice-Hall, 1989.

[Eppinger and Spector 89]
Eppinger, J., and A.Z. Spector.
Transaction Processing in Unix: A Camelot Perspective.
CamelotUnixReview 7(1):58-67, 1989.

[Eppinger et al. 89]
Eppinger, J.L., L.B. Mummert, and A.Z. Spector.
Guide to the Camelot distributed transaction facility including the Avalon language.
Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Fahlman 87] Fahlman, S.E.
Common Lisp.
Annual Review of Computer Science 2, 1987.

[Fiat et al. 88] Fiat, A., R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young.
Competitive paging algorithms.
Technical Report CMU-CS-88-196, Computer Science Department, Carnegie Mellon University, November, 1988.

[Gong and Wing 89]
Gong, C., and J.M. Wing.
Raw code, specification, and proof of the Avalon queue example.
Technical Report CMU-CS-89-172, School of Computer Science, Carnegie Mellon University, August, 1989.

[Gong and Wing 90]
Gong, C., and J.L. Wing.
A Library for Concurrent Objects and Their Proofs of Correctness.
Technical Report CMU-CS-90-151, School of Computer Science, Carnegie Mellon University, July, 1990.

[Hastings 89] Hastings, A.B.
Distributed lock management in a transaction processing environment.
Technical Report CMU-CS-89-152, School of Computer Science, Carnegie Mellon University, May, 1989.

[Herlihy 87] Herlihy,M.
Extending multiversion time-sharing protocols to exploit type information.
In *IEEE Transactions on Computers*. IEEE, April, 1987.

[Herlihy 88] Herlihy, M.P.
Impossibility and universality results for wait-free synchronization.
In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. ACM, August, 1988.
Also available as technical report CMU-CS-88-140.

[Herlihy and Tygar 87]
Herlihy, M.P., and J.D. Tygar.
How to Make Replicated Data Secure.
In *Advances in Cryptology*. August, 1987.

[Herlihy and Weihl 88]
Herlihy, M.P. and W.E. Weihl.
Hybrid concurrency control for abstract data types.
In *Proceedings of the 7th ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* . ACM, March, 1988.

[Herlihy and Wing 87a]
Herlihy, M.P. and J.M. Wing.
Avalon: language support for reliable distributed systems.
In *17th Symposium on Fault-Tolerant Computer Systems*. IEEE, July, 1987.
Also available as Technical Report CMU-CS-86-167.

[Herlihy and Wing 87b]
Herlihy,M.P., and J.M. Wing.
Axioms for concurrent objects.
In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*. ACM, January, 1987.

[Herlihy and Wing 87c]
Herlihy, M.P. and J.M. Wing.
Specifying graceful degradation in distributed systems.
Technical Report CMU-CS-87-120, Computer Science Department, Carnegie Mellon University, May, 1987.

[Herlihy and Wing 88a]
Herlihy, M. and J. Wing.
Linearizability: a correctness condition for concurrent objects.
Technical Report CMU-CS-88-120, Computer Science Department, Carnegie Mellon University, March, 1988.

[Herlihy and Wing 88b]
Herlihy, M.P. and J.M. Wing.
Reasoning about atomic objects.
In *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. University of Warwick, September, 1988.
Also available as technical report CMU-CS-87-176.

[Herlihy et al. 87] Herlihy, M.P., N.A. Lynch, M. Merritt, and W.E. Weihl.
On the correctness of orphan elimination algorithms.
In *17th Symposium on Fault-Tolerant Computer Systems*. IEEE, July, 1987.
Abbreviated version of MIT/LCS/TM-329.

[Heydon et al. 88] Heydon, A., A. Maimone, A. M. Zaremski, D. Tygar, and J. L. Wing.
Miro: A Visual Language for Specifying Security.
In *UNIX Security Workshop*. August, 1988.

[Heydon et al. 89] Heydon, A., A. Maimone, A.M. Zaremski, D. Tygar, and J.M. Wing,
Constraining Pictures with Pictures.
In *Proceedings of the IFIPS '89*, pages 157-162. August, 1989.
Also available as Technical Report CMU-CS-88-185.

[Jones and Rashid 87]
Jones, M.B. and R.F. Rashid.
Mach and Matchmaker: Kernel and language support for object-oriented distributed systems.
Technical Report CMU-CS-87-150, Computer Science Department, Carnegie Mellon University, September, 1987.
This paper also appeared in the *Proceedings of the First Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA*, September, 1986.

[Leibengood et al. 90]
Leibengood, D.L., J.G. Morrisett, S.M. Nettles, and J.M. Wing.
ML as a Basis for Distributed Object Management.
In *Standard ML Workshop*. June, 1990.

[Lerner 88]
Lerner, R.A.
Reliable servers: Design and implementation in Avalon/**C**++.
In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*. IEEE and ACM, December, 1988.
Also available as Technical Report CMU-CS-88-177.

[Maimone et al. 88]
Maimone, A., D. Tygar, and J. L. Wing.
Miro Semantics for Security.
In *Proceedings of the IEEE 1988 Workshop on Visual Languages*. October, 1988.

[Maimone, Tygar and Wing 90]
Maimone, A., D. Tygar, and J.M. Wing.
Formal Semantics for Visual Specification of Security.
Visual Languages and Visual Programming.
In Chang,
Plenum, 1990.

[Manasse et al. 88]
Manasse, M.S., L.A. McGeoch, and D.D. Sleator.
Competitive algorithms for server problems.
Technical Report CMU-CS-88-197, Computer Science Department, Carnegie Mellon University, December, 1988.

[McDonald et al. 87]
McDonald, D.B., S.E. Fahlman, and A.Z. Spector.
An efficient Common Lisp for the IBM RT PC.
Technical Report CMU-CS-87-134, Computer Science Department, Carnegie Mellon University, July, 1987.

[McGeoch and Sleator 89]
McGeoch, L.A., and D.D. Sleator.
A strongly competitive randomized paging algorithm.
Technical Report CMU-CS-89-122, School of Computer Science, Carnegie Mellon University, March, 1989.

[Mogul et al. 87] Mogul, J., R. Rashid, and M. Accetta.
A facility for rapid prototyping of networking software.
In *Proceedings of 11th ACM Symposium on Operating System Principles*. ACM, November, 1987.

[Nichols 90] Nichols, D.L.
Multiprocesing in a Network of Workstations.
PhD thesis, School of Computer Science, Carnegie Mellon University, February, 1990.

[Pausch 88] Pausch, R.
Adding input and output to the transactional model.
PhD thesis, Computer Science Department, Carnegie Mellon University, August, 1988.

[Pausch and Eppinger 89]
Pausch, R., and J.L. Eppinger.
Generating interfaces for Camelot data servers.
Guide to the Camelot distributed transaction facility, including the Avalon language.
In J.L. Eppinger, L.B. Mummert, and A.Z. Spector,
Prentice-Hall, 1989.

[Rashid 87] Rashid,R.F.
Designs for parallel architectures.
Unix Review 5(4):36-43, April, 1987.

[Rashid et al. 87] Rashid, R.F., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew.
Machine-Independent virtual memory management for paged uniprocessor and multiprocessor architectures.
Technical Report CMU-CS-87-140, Computer Science Department, Carnegie Mellon University, July, 1987.

[Sansom 88] Sansom, R.D.
Building a secure distributed computer system.
PhD thesis, Computer Science Department, Carnegie Mellon University, May, 1988.

[Satya 87] Satyanarayanan, M.
Integrating Security in a Large Distributed Environment.
Technical Report CMU-CS-87-179, CarnegieMellonUniversity, November, 1987.

[Satya et al 87] Satyanarayanan, M.
Scale and performance in a distributed file system.
In *Proceedings of 11th ACM Symposium on Operating System Principles*. ACM, November, 1987.

[Satyanarayanan 89]
Satyanarayanan, M.
A survey of distributed file systems.
Technical Report CMU-CS-89-116, School of Computer Science, Carnegie Mellon University, February, 1989.

[Satyanarayanan et al. 89]
Satyanarayanan, M., J.J. Kistler, P. Kumar, E.H. Siegel, and D.C. Steere.
Coda: A highly available file system for a distributed workstation environment.
Technical Report CMU-CS-89-165, School of Computer Science, Carnegie Mellon University, July, 1989.

[Sleator 88] Sleator, D.D. and P.F. Dietz.
Two algorithms for maintaining order in a list.
Technical Report CMU-CS-88-113, Computer Science Department, Carnegie Mellon University, September, 1988.

[Sleator et al. 88] Sleator, D., R. Tarjan, and W. Thurston.
Rotation distance, triangulations, and hyperbolic geometry.
Technical Report CMU-CS-88-108, Computer Science Department, Carnegie Mellon University, January, 1988.

[Spector 87] Spector, A.Z.
Distributed transaction processing and the Camelot system.
Distributed Operating Systems.
In Y. Paker et al.,
Springer-Verlag, 1987.
Also available as technical report CMU-CS-87-100.

[Spector 89a] Spector, A.Z.
Modular architectures for distributed and database systems.
In *Proceedings of the Eighth SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. March, 1989.

[Spector 89b] Spector, A.Z.
Introduction to Camelot.
Guide to the Camelot distributed transaction facility, including the Avalon language.
In J.L. Eppinger, L.B. Mummert, and A.Z. Spector,
Prentice-Hall, 1989.

[Spector and Swedlow 87]
Spector, A.Z. and K.R. Swedlow, eds.
Guide to the Camelot distributed transaction facility
Ver 0.92(38), Release 1 edition, Computer Science Department, Carnegie Mellon University, 1987.

[Spector and Swedlow 88]
Spector, A.Z. and K.R. Swedlow, eds.
Guide to the Camelot distributed transaction facility: release 1
0.98(51) edition, 1988.

[Spector et al. 87a]
Spector, A.Z., D. Thompson, R.F. Pausch, J.L. Eppinger,
D. Duchamp, R. Draves, D.S. Daniels, and J.J. Bloch.
Camelot: A Flexible and Efficient Distributed Transaction Processing Facility for Mach and the Internet.
In *IEEE Transactions on Computers, special issue on reliability*. IEEE, June, 1987.
Also available as technical report CMU-CS-87-129, *Camelot: A Flexible and Efficient Distributed Transaction Processing Facility for Mach and Internet—A Status Report*.

[Spector et al. 87b]
Spector, A.Z., D.S. Daniels, J.L. Eppinger, J.J. Bloch, R.P. Draves,
D.Duchamp, R. Pausch, and D.S. Thompson.
High performance distributed transaction processing in a general purpose computing environment.
In *Proceedings of the Second International Workshop on High Performance Transaction Processing Systems*. Amdahl and IBM, September, 1987.

[Spector et al. 88a]
Spector, A.Z., R. Pausch, and G. Bruell.
Camelot: A flexible, distributed transaction processing system.
In *Proceedings of Compcon 88*. IEEE and ACM, February, 1988.

[Spector et al. 88b]
Spector, A.Z., R. Pausch, and G. Bruell.
Camelot: a flexible, distributed transaction processing system.
In *Proceedings of COMPCON 88*. IEEE and ACM, February, 1988.

[Steere et al. 90] Steere, D.C., J.J. Kistler, and M. Satyanarayan.
Efficient user-level file cache management on the Sun Vnode interface.
In *Proceedings of the Summer USENIX*. USENIX, June, 1990.
Also available as technical report CMU-CS-90-126.

[Tevanian 87] Tevanian Jr., A.
Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach.
PhD thesis, Computer Science Department, Carnegie Mellon University, December, 1987.

[Tevanian and Rashid 87]
Tevanian, A. Jr. and R.F. Rashid.
Mach: A basis for future UNIX development.
Technical Report CMU-CS-87-139, Computer Science Department, Carnegie Mellon University, June, 1987.

[Tevanian et al. 87]
Tevanian, A. Jr., R.F. Rashid, D.B. Golub, D.L. Black, E. Cooper, and M.W. Young.
Mach threads and the UNIX kernel: The battle for control.
Technical Report CMU-CS-87-149, Computer Science Department, Carnegie Mellon University, August, 1987.

[Thompson 89a] Thompson, D.
The design of the Camelot recovery manager.
Guide to the Camelot distributed transaction facility including the Avalon language.
In Eppinger, J.L., L.B.Mummert, and A.Z. Spector,
Prentice-Hall, 1989.

[Thompson 89b] Thompson, D.
The design of the Camelot local log.
Guide to the Camelot distributed transaction facility including the Avalon language.
In Eppinger, J.L., L.B.Mummert, and A.Z. Spector,
Prentice-Hall, 1989.

[Tygar and Wing 87]
Tygar,J.D., and J.M. Wing.
Visual specification of security constraints.
In *1987 Workshop on Visual Languages*. IEEE and the Swedish Defense Organization, February, 1987.
Also available as Technical Report CMU-CS-87-122.

[Wendorf 87] Wendorf, J.W.
OS/application concurrency: A model.
Technical Report CMU-CS-87-153, Computer Science Department, Carnegie Mellon University, April, 1987.

[Wendorf and Tokuda 87]
J.W. Wendorf and H. Tokuda.
An interprocess communication processor: Exploiting OS/application concurrency.
Technical Report CMU-CS-87-152, Computer Science Department, Carnegie Mellon University, March, 1987.

[Wing 87a] Wing,J.M.
Writing Larch interface language specifications.
ACM Transactions on Programming Language and Systems 9(1):1-24, 1987.

[Wing 87b] Wing, J.M.
A study of twelve specifications of the library problem.
Technical Report CMU-CS-87-142, Computer Science Department, Carnegie Mellon University, July, 1987.

[Wing 87c] Wing, J.M.
A Larch specification of the library problem.
In *Proceedings of the 4th International Workshop on Software Specification and Design*. IEEE, April, 1987.
Also available as Technical Report CMU-CS-86-169.

[Wing 88a] Wing, J.M.
Specifying recoverable objects.
In *Proceedings of the 6th Annual Pacific Northwest Software Quality Conference*. PNSQC, September, 1988.
Also available as CMU technical report CMU-CS-88-170.

[Wing 88b] Wing, J.M.
Specifying Avalon objects in Larch.
Technical Report CMU-CS-88-208, Computer Science Department, Carnegie Mellon University, December, 1988.

[Wing 88c] Wing, J.M.
A study of 12 specifications of the library problem.
IEEE Software July:66-76, 1988.

[Wing 89a] Wing, J.M.
Verifying atomic data types.
Technical Report CMU-CS-89-168, School of Computer Science, Carnegie Mellon University, July, 1989.

[Wing 89b] Wing, J.M.
Specifying Avalon objects in Larch.
In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Barcelona, Spain*. March, 1989.

[Wing 89c] Wing, J.M.
What is a formal method?.
Technical Report CMU-CS-89-200, School of Computer Science, Carnegie Mellon University, November, 1989.

[Wing 90] Wing, J.M.
A specifier's introduction to formal methods.
Technical Report CMU-CS-90-136, School of Computer Science, Carnegie Mellon University, May, 1990.
Also in *IEEE Computer*, September, 1990.

[Wing and Gong 89]
Wing, J.M., and C. Gong.
Machine-assisted proofs of properties of Avalon programs.
Technical Report CMU-CS-89-171, School of Computer Science, Carnegie Mellon University, August, 1989.

[Wing and Gong 90a]
Wing, J.M. and C. Gong.
Experience with the Larch prover.
In *Proceedings of the ACM Workshop on Formal Methods in Software Development*. ACM, May, 1990.

[Wing and Gong 90b]
Wing, J.L., and D. Gong.
A Simulator for Concurrent Objects.
Technical Report CMU-CS-90-150, School of Computer Science, Carnegie Mellon University, July, 1990.

[Wing et al. 88] Wing, J., M. Herlihy, S. Clamen, D. Detlefs, K. Kietzke, R. Lerner, and S. Ling.
The Avalon/C^{++} programming language (version 0).
Technical Report CMU-CS-88-209, Computer Science Department, Carnegie Mellon University, December, 1988.

[Yee et al. 88] Yee, B.S., J.D. Tygar, and A.Z. Spector.
Strongbox: A self-securing protection system for distributed programs.
Technical Report CMU-CS-87-184, Computer Science Department, Carnegie Mellon University, January, 1988.

[Young 89] Young, M.W.
Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.
PhD thesis, Computer Science Department, Carnegie Mellon University, November, 1989.

4. RESEARCH IN PROGRAMMING ENVIRONMENTS

The increasing sophistication of today's software creates a need for greater sophistication of the programming environments used in software development efforts. Future environments will have to support large, complex development projects. Such projects might involve many cooperating but widely-distributed participants who have diverse requirements for managing, propagating, and communicating information. These next-generation environments will have to offer three crucial features:

- The ability to evolve incrementally as developers integrate new tools and data into existing systems
- Communication support that allows developers and tools to coordinate their activities
- Automation of the user interface to enhance efficiency.

In addition, the high cost of software development mandates that these environments be automatically generated, rather than hand-crafted.

At Carnegie Mellon, our research in programming environments continues to build on the Gandalf system, an environment generator. Gandalf environments maintain knowledge about the project at hand in a set of databases and can shoulder many of the burdens (such as integrating programming tools with system development support) previously left to the programmer.

Gandalf users are divided into three classes: *kernel implementors*, *environment designers*, and *end users*. Kernel implementors are our researchers here at Carnegie Mellon who design and build Gandalf. Environment designers are those people who build software development environments using the Gandalf tools. Since Gandalf is *bootstrapped* (successive versions of Gandalf are built using Gandalf tools), the kernel implementors are also environment designers. An end user is anyone who works in an environment generated with Gandalf. To the end user, Gandalf itself is transparent.

In response to the needs identified above, our research has focussed on these four areas:

Data Transformations

Our work on data transformations responds to the difficulty of incorporating new release versions of software into existing environments; as software becomes increasingly complex, converting existing data structures to be compatible with a new software release becomes a more difficult and time-consuming procedure. We explored ways of automating this conversion process.

Views for Tools

One goal of our research has been to support a style of programming where several tools can be written separately and later combined into a complete program. This is in stark contrast to current programming environments, which depend upon a central database or the good will of programmers to maintain system consistency and tool compatibility.

Communication Support

As software complexity increases and larger numbers of programmers work on a single project, issues of communication and control become larger. Our research in communication support explored three key areas that previous programming environments had failed to address successfully:

- Database segmentation
- Concurrency
- Configuration management

Expertise and Tolerance

Today's computers and software are more powerful than ever before, but state-of-the-art systems still rely on the *user's* knowledge of the computer to facilitate communication instead of allowing the computer to take over that burden. We investigated ways of making the interfaces to programming environments more knowledgeable of the user's preferences and more tolerant of his errors. A significant problem with such an interface is the cost (in terms of efficiency) of automaticity and tolerance, especially if the consequent gain in productivity is not great. Our research on tolerant user interfaces focused on trying to maximize the efficiency of these interfaces.

4.1. Data Transformations

A serious problem for programming environments is that information created and maintained by a programming environment becomes invalid when the environment is replaced by a new release. This problem is not unique to programming environments; it also affects many other types of programs including database systems and operating systems. For example, in the widespread conversion from Version 4.1 of BSD UNIX to Version 4.2, file directory structures created under 4.1 were incompatible with 4.2 and therefore required conversion. To install a new release of an environment or system requires, at the very least, design and construction of a conversion process for existing persistent data, and work must partially halt while conversion takes place. Users are burdened with a period of instability and loss of functionality in the case of inadequate conversions. Consequently, users and environment designers are faced with a dilemma: stability can be achieved by ignoring successive releases, in which case the environment or system will not meet the evolving needs of its users; or change can be allowed, at the cost of a time consuming process of conversion.

Structure-oriented environments such as those generated by Gandalf are a class of programming environments for which these problems are particularly severe. A structure-oriented environment is typically generated from a formal description, including a *grammar*. One of the purposes of the grammar is to define the structure of databases created with such an environment. When the grammar is changed in any but trivial ways, existing databases, structured according to the old grammar, may not be compatible with the new grammar, thus preventing the new environment from using old data. At early stages of environment prototyping, it may be feasible to simply discard the old databases. However, in practical settings where users have come to depend on the information and programs created with the old environment, a sudden announcement that existing databases are no longer valid will not be acceptable.

Our work at Carnegie Mellon has shown how automatic converters can be generated in terms of an environment designer's changes to the grammars of structure-oriented environments. We designed and implemented an environment called *TransformGen*, in which an environment designer can make structured changes to the grammar of an environment. The output of TransformGen is a new grammar together with a *transformer*, which takes instance of database trees built under the old grammar and automatically converts them to instances of database trees that are legal under the new grammar [Staudt et al. 88].

While our techniques were developed specifically to solve problems of grammar evolution for structure-oriented environments, many of the results carry over to other systems. In particular, our experience indicates that there are three essential ingredients to a successful approach to maintenance based on structural transformation. First, the objects to be transformed must be represented in a structured form as described by some formal notation. Second, it is important to provide an environment in which monitored changes can be made to this notation. Third, any resulting transformation scheme must be extensible along two dimensions: it must be possible to augment the repertoire of transformations automatically handled by the transformer as new classes of transformation become better understood, and it must be possible for the person who is making the changes to augment the automatic mechanisms to handle special cases.

The results of pursuing this approach, at least within the domain of structure-oriented environments, have been encouraging. We have been able to make substantial improvements to existing environments that would have been infeasible using the manual, ad hoc techniques available before TransformGen. The generator for structural transformations is a powerful tool that can be built relatively easily by extending existing environment generators.

Gandalf now incorporates TransformGen in AloeGen, a tool used to create and maintain environment grammar descriptions. AloeGen monitors changes and automatically transforms data from previous versions. Previously, any nontrivial change in the database grammar's syntax would invalidate trees created under the prior grammar.

4.2. Views

Most programming environments exhibit one of two strategies for incorporating tools. The "toolbox approach" describes those environments that comprise a loosely organized toolset on top of a host operating system. Tools exist more or less in isolation, and the primary burden of maintaining system consistency lies with the environment's users. Since the environment itself has no knowledge of tools or tasks, it can provide little help in supporting the management and communication requirements in a large project.

Another approach has been that of integrated environments, whose cooperating tools share a common database and an environment kernel that mediates interactions between user, tool, and database. A problem with this approach is that tool interdependence extends all the way down to the shared data structures, so that data representations for one tool depend upon the formats that other tools use. Building and modifying tools can present a special challenge when they require data structures incompatible with those in the existing database. Often, all work is halted until a central database can be reconfigured.

The Gandalf project studied how to integrate tools by specifying "views." The process requires a common database of shared structures. The main difficulty is defining a data format that satisfies all the tools. The traditional approach, defining the data structure then assembling the tools, is self-limiting since further evolution is restricted to that structure. Instead, the system should be able to determine and adapt that data structure to what is expected by the tools. In our model, each tool defines a *view* into the common database according to what it wants to see in terms of data and operations on that data. The system then *synthesizes* the data format from the collection of views for the tools to be integrated.

The goal of our research in views was to produce a concrete, coherent language with support for views-style programming. Toward this end we designed and developed a language (Janus) for views. Due to uncertainties about the language that developed as the research progressed, we did not go on to develop a compiler. Janus started as an object-oriented language, but in some ways a conventional language with abstract types now seems like a more appropriate foundation for views programming.

Conventional languages make a strong distinction between code and data; much of the appeal of the object-oriented style was lost when we found ourselves making precisely the same distinction with Janus, as well. The abstraction boundary associated with an abstract data type partitions the code associated with it in exactly the desired way:

1. Code that *implements* the data type. This code sees the concrete representation of the type and is responsible for maintaining the invariants of the abstraction in terms of the representation.
2. Code that *uses* the data type in other computations.

The code that implements a type would be rewritten when a merged representation is substituted for the original abstract data type (ADT). Code that uses the abstract type should be unaffected. This is essentially what we achieved with our object-oriented Janus design; the ADT style would simply make it more natural by explicitly supporting *procedures* that operate on abstract data.

But there are difficulties with the ADT approach, as well. One of the main differences between conventional ADTs and objects is that objects encapsulate both private storage and code for operating on that storage, while ADTs have a single, central copy of the code. This means that all instances of an ADT will have exactly the same runtime representation, which is both good and bad:

- The advantage of ADTs is that code that can see their representation may directly manipulate two or more instances at the same time. In object oriented programming, the fields of exactly *one* object are available at any given time Other objects may be accessed only abstractly.
- The advantage of objects is that there may be many representations of the same class at runtime. Each instance knows which representation it is using, and encapsulates code that operates correctly on that representation.

The concept of views remains a powerful one. However, because of these difficulties with the Janus language, a new language must be developed before the power of views can be fully realized [Habermann et al. 88].

4.3. Communication Support

Structure editors are traditionally viewed as tools restricted to programming-in-the-small. At the other end of the spectrum, environments for the programming-in-the-large domain (i.e. numerous large program modules) typically view a software database as a set of black boxes with a narrow interface providing no knowledge of the internals.

By providing the proper database support, environments generated by the Gandalf system can address programming-in-the-large as well as programming-in-the-many (i.e. numerous programmers on a single project) without the loss of the fine grain information available in the knowledge-based environments currently generated.

In providing this support, some of the issues we examined were:

- *Gandalf's previous restriction of storing all information in a single monolithic database.* We wanted to maintain all the knowledge stored in a Gandalf environment, but not as a single structure. The monolithic database often proved cumbersome to work with. The database generated by the improved Gandalf system produces a set of integrated software databases in which all information is structurally decomposed for storage and accessed via a single uniform interface. This assists not only the end-user, but also the environment designer by allowing the decomposition of their specifications.
- *Concurrency and its associated management tasks.* The addition of a seg-

mented database will, by providing quicker access to stored information, stimulate the desire for multiple users within that database (task decomposition). To enhance our segmented architecture, we are implementing a persistent transaction mechanism to provide both *controlled* concurrency of multiple users and the ability to start new child transactions for experimental workspaces. Along with implementing the transactions, our design includes a copy-on-write facility to minimize the proliferation of files that can occur with the sharing of files across transactions.

- *Configuration management.* Large software projects complicate the management tasks of tracking versions and configurations. A generated environment needs to support these tasks, but it should be left to the environment designers to specify their needs in a generated environment. Gandalf provides a basic default model for configuration management, but it is easily customizable. Many other development environments have erred by "hardwiring" these policies into unmodifiable code.

4.3.1. Segmented Architecture

For interactive structure-oriented programming environments such as those generated by the Gandalf System, it is infeasible to have a single database server providing database access for multiple user processes, for two reasons:

- The computer resources required to extract the textual representation of information from the database
- The high bandwidth requirements for passing this information between the database and the user process.

These suggest that each user process must have direct access to the database. However, with very large software databases it may also be impractical to load an entire database into a user process space. A compromise between these two conflicting efficiency considerations is to segment the database contents and allow user processes to access only small segments of the database directly.

Providing facilities for database segmentation allows an environment to be used in real programming-in-the-large applications. It reflects a desire to provide good system performance for programming-in-the-small activities within a large software database.

During this funding period, we have designed and implemented a scaled-up version of the Gandalf System to support full scale software development projects. This version meets our objectives of modular database grammars, segmentation of the software database contents, allowing multiple users to concurrently access the software database, and still retain the small grain database integration present in earlier Gandalf environments. Grammar modularity, segmentation, and concurrency are achieved by segmenting the software database at grammar boundaries and applying concurrency at the segment level of granularity. Small grain database integration is achieved through unification of the two levels in the database: the internal segment level and the atomic segment level. This is accomplished by having the segment node, at the internal segment level, provide the structural relationships between segments at the atomic segment level. The cooperation of the internal segment database manager and the atomic level database manager bridges the boundaries around segments at the atomic segment level.

Access control was found to be important in supporting the reserve/deposit semantics required in software transactions. Due to the hierarchical, large-grained, and long-lived nature of software transactions, it is important to be able to delimit portions of the software database for a semantically related collection of changes by a group of one or more persons. Access control mechanisms for reading and modifying segments provides the Gandalf System with reserve/deposit semantics [Krueger and Bogliolo 88].

Support for segmented architecture has been incorporated into Gandalf and used successfully for about a year.

4.3.2. Concurrency

A software development environment must provide controlled concurrent access to the environment database in order to support a team development effort. Such access control must include features similar to those in conventional database systems, such as read/write locks. However, due to fundamental differences in transaction characteristics, standard solutions to the concurrency problem found in the database literature may not be directly applicable to software databases. Unlike transactions in conventional database systems, concurrent transactions in a software database may have life spans on the order of days or weeks and may involve kilobytes of information. Because of this large transaction size, it is important for the user to know in advance that a transaction will not fail due to factors external to the transaction itself. For example, after programming for a month, most developers will be unwilling to "abort the transaction and try again" because a lock required for subsequent development is unavailable.

Providing facilities for controlled concurrent access to a software database allows an environment designer to support programming-in-the-many. For example, access control tests can be added to read/write locks in the database to control which persons and groups can read and modify information.

Our work on concurrent transactions at Carnegie Mellon assumes the following about a typical transaction for one or more programmers operating in a software database:

- The transaction will be long term (on the order of days or weeks)
- It will involve a large collection of semantically related extensions and modifications.

Although transactions of this type can be viewed as a long sequence of smaller transactions, the top level transaction will only be able to *commit* if all of the smaller subtransactions succeed. For example, if a procedure declaration is modified to have an extra parameter, then the transaction is not complete until *all* of the procedure use sites have been modified to reflect syntax and semantics of the additional parameter.

Our notion of transaction involves the programmer as much as the operations on the database. In order to change the procedure use sites in our example, the programmer must understand the intended semantics of the change in each use site context and issue the appropriate commands to modify the database accordingly.

In order to avoid failed transactions due to locking or other external conflicts, we have developed the *reserve* operation. The reserve operation delimits portions of the database for a semantically related collection of changes by a group of one or more persons. At first glance it appears that a write lock is sufficient for the reserve. However, there are at least two fundamental differences between the reserve and write lock semantics. The first is that reservation by a group should not necessarily preclude others from viewing reserved information, only from writing or acquiring read or write locks. The second difference comes from having a group of persons involved in the reservation. Although a group has the right to modify its reserved portion of the database, it still only makes sense to allow a single programmer to gain a write lock on any given piece of information at any one time. This illustrates that the semantics of the reserve operation is to limit the modification and access privileges on a portion of the database to members of a group.

The current Gandalf System generates single-user environments. It provides no control for concurrent database access. We have, however, designed and built a kernel version which includes support for concurrent transactions. It is undergoing testing prior to release.

4.3.3. Configuration Management

For a software development environment to handle a large, complex software project effectively, the environment must have mechanisms for source code control and configuration management. A common method of addressing this problem uses a source control system such as RCS (Revision Control System) as a baseline. Configuration management is superimposed upon the low level version control system by describing a *system variant* as a selection thread through the sets of *component* versions. In the Gandalf project we explored the inverse approach, building version control on top of configuration management.

Gandalf uses three levels to manage a software system. At the lowest level in our model we maintain a hierarchically structured collection of source code components. This structured collection consists of one or more variants of a software system. Each variant represents a complete instance of the software system and is comprised of a subset of the components in the collection, such as a system variant to run under the X windowing system versus a system variant to run under the NeWS windowing system. This approach emphasizes the composition of systems as opposed to assembling system variants out of components organized with a version control system at the lowest level.

The next higher level provides version control. This level maintains the evolution of the system as a set of *revisions*. As development proceeds, modifications of software components are tracked by the creation of additional revisions. Each revision contains a full set of variants in contrast to systems that maintain revisions of single components.

At the top level, we embed version control into nested *long-term transactions* to handle the issues of active software development by multiple users. This mechanism defines a recursive structure that provides encapsulated workspaces for software development and the ability to divide the development process into subtasks. All levels of the transaction hierarchy include the capabilities of the versioning system, providing the developer with a safe environment for experimental work [Miller et al. 89].

4.4. Expertise and Tolerance

Computer technology has reached a stage where automatic customization of user interfaces is feasible. In the past, the user has been the computer's servant, feeding data upon demand, and strictly following the often arcane rules set down by the application's programmer. But this no longer needs to be the case. Indeed, our research has striven to make the computer be subservient to the user. The computer should learn to understand a user's requests, rather than force the user to speak its language.

Towards this goal, we identified three properties that an intelligent user interface should exhibit:

- *Automation*. The interface should automate as much of the user's task as is feasible.
- *Tolerant command interpretations*. the interface should be tolerant of the styles of individual users.
- *Active help*. When necessary, the interface should provide help and explanations actively, and in terms that are adapted to the individual user.

At Carnegie Mellon, our research on tolerant user interfaces has concentrated on designing an object-oriented architecture, that will support the definition of heuristics. These heuristics should allow a user interface to adapt to individual users based on domain, context, and historical knowledge.

We have designed and built a prototype architecture which allows the environment designer to define three different kinds of procedural objects required to support heuristics: heuristics, loggers, and success-failure criteria. *Heuristics* perform the actions that actually modify the user interface. *Loggers* collect and store those parts of the history which the heuristics require. *Success-failure criteria* monitor the user's actions after a heuristic has been applied to determine whether the heuristic's actions were acceptable to the user. Heuristics and loggers can be attached to node types, error types, and command types. Success-failure criteria can be attached to node instances, error types, and command types. The environment designer initially defines where heuristics and loggers should be attached. As the user uses the system, those heuristics whose performance falls below some pre-defined standard are automatically detached from their objects. Therefore, after a brief training period, only those heuristics which are actually useful for the user will remain attached. Those heuristics which are not useful will no longer be attached and thus will not detract from the performance of the ALOE (a language oriented editor) [Lerner 89a].

We have designed and built a prototype of the heuristics architecture, which has been tested successfully.

4.5. Changes to Tools

4.5.1. Language for Specifying Semantics

ARL (Action Routine Language) is our language for specifying semantics. Its syntax augments the user’s ability to write action routines and improves the routines’ screen representations.

Although ARL was an excellent tool for specifying semantics, it handled string manipulation poorly and programmers had to manipulate strings via their own **C** code. Responding to this problem, we added a string library to the ARL environment. The library contains routines that help produce complex strings such as error messages. Previously, functions relied on sequential string concatenation that was prone to "memory leaks" because it allocated memory without subsequently de-allocating it properly.

We have also removed the use of *cursors*, which are pointers to nodes, as a part of the language. We found users often had problems distinguishing when to use a node variable and when to use cursors. The use of cursors was needed to protect nodes which were being referenced to avoid unwanted side effects of certain constructive/destructive commands. The protection has now been incorporated into the commands themselves, so that an environment designer only uses node variables. The ARL environment includes the appropriate transformations to automatically convert earlier versions of ARL trees. We have noted a few instances where the semantics cannot be automatically converted and will identify these sites requiring further input by the designer.

4.5.2. LexGen

Responding to requests for additional facilities, we have generated a second version of the LexGen editor and improved our distribution mechanisms for ALOEs. The LexGen editor is used to specify scanning routines that validate user input. The new LexGen includes improved error checking and a mechanism that can define named macros, thus allowing named regular expressions. These changes have been documented in user manuals.

4.5.3. Input Parsing

We have been integrating the use of parsing input into the lines of a command editor. The use of incremental parsing allows a more natural "infix" style of input for longer expressions. Incomplete information is handled via a variety of methods.

- The user can input the name of a *metanode* as a placeholder so that the rest of the expression can be typed out. The user would later fill in the metanode.
- The user can use a generic "$$" token for specifying whatever metanode would be appropriate in the context of an expression instead of naming the metanode explicitly, in unambiguous cases.
- We have created a special version of ‘yacc’ that can handle some "error repair" automatically. This is a public domain version, allowing us to distribute the modified form. Without additional input from the environment designer, there are many cases where the editor can automatically construct metanodes to validate input that would have previously generated a syntax error. For the Pascal environment, we found the system can repair 86% of the cases with metanodes as the lookahead that would have generated a syntax error.

4.6. Bibliography

[Habermann et al. 88]
Habermann, A.N., C. Krueger, B. Pierce, B. Staudt, and J. Wenn.
Programming with views.
Technical Report CMU-CS-87-177, Carnegie Mellon University Computer Science Department, January, 1988.

[Krueger 89] Krueger, C.W.
Models of reuse in software engineering.
Technical Report CMU-CS-89-188, School of Computer Science, Carnegie Mellon University, December, 1989.

[Krueger and Bogliolo 88]
Krueger, C. and A. Bogliolo.
Scaling up: segmentation and concurrency in large software databases.
Technical Report CMU-CS-87-178, Carnegie Mellon University Computer Science Department, February, 1988.

[Krueger et al. 89] Krueger, C.W., B.J. Staudt, and A.N. Habermann.
Scaling up integrated software development environment databases.
In *Proceedings of the 1989 ACM SIGMOD Workshop on Software CAD Databases*. ACM, February, 1989.

[Lerner 89a] Lerner, B.S.
Automated customization of user interfaces.
PhD thesis, School of Computer Science, Carnegie Mellon University, September, 1989.
Also available as technical report CMU-CS-89-178.

[Lerner 89b] Lerner, B.S.
Building automatically customizable user interfaces: Position paper on modeling for intelligent user interfaces.
In *A New Generation of Intelligent Interfaces, IJCAI '89 Workshop*. IJCAI, August, 1989.

[Miller 90] Miller, D.B.
Structure editors for systems development.
In *Proceedings of CHI'90 Workshop on Structure Editors*. , April, 1990.

[Miller et al. 89] Miller, D.B., R.G. Stockton, and C.W. Krueger.
An inverted approach to configuration management.
In *Proceedings of the 2nd International Workshop on Software Configuration Management*. Sponsored jointly by ACM & IEEE, October, 1989.

[Staudt et al. 88] Staudt, B., C. Krueger, and D. Garlan.
TransformGen: Automating the maintenance of structure-oriented environments.
Technical Report CMU-CS-88-186, Carnegie Mellon University Computer Science Department, November, 1988.

5. RESEARCH IN REASONING ABOUT PROGRAMS

The central problem of software engineering is the development and maintenance of demonstrably correct programs. In order to accomplish this goal, a wide variety of approaches to the problem have been developed. These range from what might be called ‘‘software management techniques,’’ devoted primarily to addressing the practical questions of how best to manage a software development project in order to improve the quality of programs, to fundamental research programs that seek to develop a rigorous body of theory on which software development methodologies may be based. These two approaches (among many others) are not opposed to one another, but rather address two equally important aspects of the problem: on the one hand, the question of what can be achieved now, and on the other, the question of what might be achievable at some point in the future. Research into the development of reliable and maintainable software is an ongoing dialogue between workers in these two areas, with fundamental research results continually being incorporated into the mainstream of software development, and with the problems of practical software development providing the framework in which fundamental research is being conducted.

Our research is devoted to the development of a comprehensive and rigorous mathematical foundation for programming. This foundation will support reasoning about the correctness of language implementations, programs, and software development methods, and will also be vital to the coherent development of languages, implementations, and program tools. In broad terms, our research focuses on three main areas: mathematical theories underlying parallelism, development of advanced type systems, and applications of mathematical logic in program development. Specific activities include:

- Use of formal semantics and type theory to design and prototype advanced programming languages
- Development of formal systems for analyzing and synthesizing programs
- Design of software tools for deriving and verifying program correctness.

We are linked with the Ergo project, which has developed the Ergo Support System (ESS) [Lee et al. 88]. The ESS is an integrated environment for experimenting with advanced programming methodologies based on formal proof techniques. Research on reasoning about programs provides the necessary theoretical underpinnings for continued development of ESS tools and components. Conversely, the ESS provides a test-bed and environment for experimentation and rapid prototyping of new theoretical ideas.

5.1. Semantic foundations of parallel programming

The first focus of our research is the semantic foundations of parallel programming. Our main research results in this area can be organized into two categories: The development of a proof methodology for analyzing deadlocks and correctness in parallel programs, and the development of mathematical models for reasoning about intensional properties, such as efficiency, of parallel programs.

A proof methodology for parallel programs

The well-known notion of ‘‘weakest precondition’’ and the closely related ‘‘strongest postcondition’’ represent fundamental tools in the theory and practice of proving program correctness. While these ideas are well understood for sequential programs, until recently they have not been adequately adapted to parallel programming languages. In the parallel setting there are many complicating features that make it impossible to adapt naively ideas that work in the sequential domain. Such features include the potential for deadlock and the often intricate ways in which program components can interfere with one another.

We have successfully developed a new axiomatic method for parallel program proofs. Its principal innovation lies in employing tree-structured assertions and incorporating a generalized form of ‘‘weakest preconditions’’ cleanly and elegantly into the parallel setting.

Building upon this result, we have developed a proof methodology that guides one through a rigorous ‘‘development’’ of the proof of a parallel program’s correctness. This method makes it relatively easy to track cleanly the potential for deadlock and the interferences between program pieces and represents a significant advance over earlier parallel proof methodologies that typically force one to prove correctness separately from proving deadlock-freedom. We believe that this new methodology allows more straightforward program proofs and is conceptually easier to use than previous techniques.

Today, we continue investigating the utility of this method as the basis for several program-development methodologies, and have considered problems posed by real-world applications. For example, in a separate but related line of research, we have been investigating ways to prove the absence of network deadlock by reducing the problem to small subnetwork deadlock. This reduction decreases the size of the combinatorial explosion normally incurred when attempting to analyze large systems [Brookes and Roscoe 89].

Mathematical models of parallel programs

We are also developing techniques for reasoning about intensional properties, such as efficiency, of parallel programs. Eventually we plan to use these techniques as the basis for designing and implementing automated tools that can assist in general reasoning about parallel programs. We view our work here as a novel departure, in that most traditional semantic models focus on purely extensional aspects of program behavior, such as input-output behavior and partial or total correctness. By building a more finely structured model that accurately represents intensional information about program behavior (such as a computation strategy suitably formulated as a mathematical object), we will be able to provide a semantics that supports reasoning about how efficiently a program computes its results. We believe that such a fundamental investigation is necessary to understand the potential for exploiting parallelism in programming language design and implementation.

Our work has progressed well, and our main result is a new mathematical model of parallel programs that is suitable for reasoning about intensional behavior of deterministic parallel functional programs [Brookes and Geva 89]. One possible outcome of this work might be the design of a powerful, yet clean and elegant parallel programming language, in which the programmer can employ parallelism conveniently wherever its use can result in greater efficiency.

A key feature of the model is the mathematical representation of the ''computation strategy'' used by an algorithm to compute its results. The objects of this model are called ''parallel algorithms,'' to distinguish them from the functions they compute and to emphasize the fact that they truly embody parallelism [Brookes and Geva 90]. (This model arises as a generalization of Berry and Curien's earlier ideas on ''sequential algorithms.'') We believe that our new construction may lead to a coherent semantical account of parallelism that sheds new light on several issues: the relationships among various alternative models of parallelism, as well as among alternative semantic models, and the relative expressive powers of various parallel primitives. In the course of demonstrating that our model is sensible, we discovered a new ordering on algorithms that is based on the notion of ''strictness.'' In the new ordering, the conventional extensional ordering on functions generalizes naturally to the intensional setting. This is an encouraging sign that previously developed methods for reasoning about extensional properties can be adapted to reasoning about intensional properties.

5.2. Development of advanced type systems

Although this early work on type systems in programming languages led to significant advances in software engineering, their success must still be described as limited. Years of research, however, have changed the terms of the debate about the value and role of types considerably. Languages incorporating notions such as *type polymorphism*, *type inclusion*, and *type inference*, which ameliorate or eliminate many of the restrictions of early type systems, are becoming more widely used and implemented. At the same time, the mathematical theory of types has been elaborated to a considerable extent, leading to new applications of types such as the use of types as specifications, and to greater understanding of the deep structure of programming languages. At present there is a considerable body of research devoted to the applications of types in programming, ranging from improved type systems for functional and object-oriented languages, to the applications of types in formal program development.

In our work on reasoning about programs, we have focused on the development of two specific kinds of type systems. The first kind is called an ''intersection type discipline,'' which is being studied in the context of a new language design called Forsythe. The second kind involves a mechanism called ''stratified polymorphism.''

Intersections types and the Forsythe language

As an example of what intersection types mean in practice, consider a language in which the operators ‘‘+’’ and ‘‘*’’ act upon both integers and reals. Using intersection types, the user might define a procedure

```
procedure poly (x); x*x+2*x+1
```

that can act upon either integers or reals. In contrast to the typical ‘‘generic’’ procedure capability (as in, for example, Ada), such a procedure is *guaranteed* by the language design to interact cleanly with the implicit conversion from integers to reals, so that the same result occurs whether poly is applied before or after converting an integer to a real.

This is, of course, an extremely simple example, but it illustrates the point that such guarantees — which we believe to be crucial for any sound approach to software engineering — can be obtained by working from the mathematical semantics of the type system. In contrast, ad hoc approaches lead to languages with many inconsistencies and ambiguities, and for which such guarantees lack the force of mathematical proof.

We have been successful in defining a semantics, based on category theory, for intersection types. This semantics incorporates such guarantees and is sufficiently general to be applicable both to functional programming languages and to Algol-like languages that combine functional capabilities with imperative features [Reynolds 87].

With the semantics of an intersection type discipline in hand, we were then able to design ‘‘Idealized Algol,’’ a language based upon a semantic model of Algol-like languages that emphasizes their close connection with the lambda calculus. The result has been a substantial simplification and generalization of the language, now named ‘‘Forsythe.’’ One benefit is that the programmer is allowed to define his own declarations (e.g., for unusually shaped arrays) straightforwardly. Another is that the language now supports object-oriented programming in a meaningful way, including the notion of attribute inheritance. A detailed description of Forsythe is given in [Reynolds 88] (which also contains 225 lines of programming examples, including an object-oriented program for finding paths in a directed graph).

Recently, we have shown that the semantics of Forsythe is unambiguous. This question arises because the type structure is sufficiently rich that there can be more than one proof that a program phrase has a particular type. Since the formal semantics is defined by induction on such proofs, one must show that the language is ‘‘coherent,’’ in other words, that distinct proofs of the same typing always give the same semantics. We have proved the coherence of a class of intersection-typed languages, including Forsythe, in [Reynolds 91].

We have also investigated how to restrict the syntax of Forsythe in order to make aliasing between variables and, more generally, interference between procedural side-effects, detectable during compilation. Such a restriction is a vital first step in extending most any imperative language to parallel processing, since it is necessary if the compiler is to detect when parallel processes can interfere with one another. A sufficient system of restrictions has been found and reported in [Reynolds 89]. It may be that these restrictions are too draconian, and thus we are refining this system further before imposing them on a practical language.

Finally, we have discovered that type checking for intersection-type languages, such a Forsythe, is PSPACE-hard. (This result has not yet been published.) This implies that there exist programs for which type checking will be very inefficient, but not necessarily that such programs will arise in practice. Nevertheless, it is a clear warning that the Forsythe type checker will have to be written very carefully to avoid unnecessary inefficiency.

These various technical results about intersection types and Forsythe constitute significant technical contributions. Moreover, they represent a new, disciplined, and sound approach to language design, implementation, and analysis which may serve as a much better foundation for software development methodologies.

Stratified polymorphism

Higher-order logic has long served as an extremely powerful tool in the study of logic and mathematics. We have been exploring how to use higher-order logic as a unifying framework for program synthesis, program analysis, and computational complexity. Our work has successfully established several fundamental connections between higher-order logics, on one hand, and the logics of programs and computational complexity, on the other. Somewhat surprisingly, these investigations have disclosed relevant work from the 1910's and 1950's.

Our main innovation is the development of a notion of a ''stratified polymorphic type discipline'' [Leivant 89a]. The key idea here is the use of a spectrum of type disciplines based on type quantification with stratified levels, ranging from the so-called ''parametric polymorphism'' of ML and the full quantification of the second-order polymorphic lambda calculus. Stratified polymorphism has an attractive, straightforward semantics, and thus has the potential for offering new approaches to type inference without sacrificing useful expressive power.

5.3. Applications of mathematical logic in programming

A natural extension to the research in types is the search for applications of higher-order logic in programming. The principle problem is as follows: ''Given a formal proof that a function satisfies its specifications, find an algorithm for computing the function.''

One approach to this problem is the so-called Curry-Howard isomorphism, which makes a precise analogy between formulas and proofs on the one hand, and types and programs on the other. In other words, for certain kinds of logical systems, formulas can be viewed as program specifications, and proofs of the formulas as programs that satisfy the specifications.

We have developed a generalization of this analogy to the more powerful, higher-order logical systems and shown how this may be systematically used in deriving correct programs [Leivant 89b]. This generalization has an advantage over previous formulations in its great clarity and generality. For example, it does not depend on the presence of specific data types, but only on the logical form of their specifications. This leads to an elegance and simplicity which may be more conducive to automated support in an advanced environment such as the Ergo Support System.

5.4. Bibliography

[Brookes and Geva 89]
Brookes, S.D., and S. Geva.
Parallel exponentiation of concrete data structures.
Technical Report CMU-CS-89-206, School of Computer Science, Carnegie Mellon University, December, 1989.

[Brookes and Geva 90]
Brookes, S.D. and S. Geva.
Towards a theory of parallel algorithms on concrete data structures.
In *Proceedings of the International Workshop on Semantics for Concurrency*. Springer Verlag, July, 1990.

[Brookes and Roscoe 89]
Brookes, S.D., and A.W. Roscoe.
Deadlock analysis in networks of communicating processes.
Technical Report CMU-CS-89-161, School of Computer Science, Carnegie Mellon University, June, 1989.

[Brookes and Roscoe 90]
Brookes, S.D., and A.W. Roscoe.
Deadlock analysis in networks of communicating processes.
Distributed Computing , 1990.
Also available as Technical Report CMU-CS-89-161.

[Geva 89]
Geva, S.
Parallel Exponentiation of Concrete Data Structures.
Technical Report CMU-CS-89-206, School of Computer Science, Carnegie Mellon University, December, 1989.

[Lee et al. 88]
Lee, P., F. Pfenning, G.J. Rollins, and W.S. Scherlis.
The Ergo Support System: An integrated set of tools for prototyping integrated environments.
In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 25-34. ACM, November, 1988.

[Leivant 87]
Leivant, D.
Characterization of complexity classes in higher order logic.
In *Proceedings of the Second Annual Conference in Complexity Theory*. IEEE, June, 1987.
Submission invited to a special issue of the Journal of Computer and System Sciences.

[Leivant 89a]
Leivant, D.
Stratified polymorphism.
In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. IEEE Computer Society, 1989.

[Leivant 89b] Leivant, D.
Contracting proofs to programs.
Technical Report CMU-CS-89-170, School of Computer Science, Carnegie Mellon University, July, 1989.

[Leivant 89c] Leivant, D.
Inductive definitions over finite structures.
Technical Report CMU-CS-89-153, School of Computer Science, Carnegie Mellon University, June, 1989.

[Leivant and Fernando 87]
Leivant, D. and T. Fernando.
Skinny and fleshy failures of relative completeness.
In *14th Symposium on Principles of Programming Languages*. ACM, January, 1987.
Also submitted to the Journal of the ACM.

[Pleban and Lee 88]
Pleban, U. and P. Lee.
An automatically generated, realistic compiler for an imperative programming language.
In *Proceedings of the Conference on Programming Language Design and Implementation*. SIGPLAN, June, 1988.

[Reynolds 87] Reynolds, J.C.
Conjunctive types and Algol-like languages.
In *Procededings of the Symposium on Logic in Computer Science*. June, 1987.
Abstract of invited lecture.

[Reynolds 88] Reynolds, J.C.
Preliminary design of the programming language Forsythe.
Technical Report CMU-CS-88-159, Carnegie Mellon University Computer Science Department, June, 1988.

[Reynolds 89] Reynolds, J.C.
Syntactic control of interference: Part 2.
Technical Report CMU-CS-89-130, School of Computer Science, Carnegie Mellon University, April, 1989.

[Reynolds 91] Reynolds, J.C.
The coherence of languages with intersection types.
In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*. Springer-Verlag, September, 1991.

[Reynolds and Plotkin 88]
Reynolds, J.C. and G.D. Plotkin.
On functors expressible in the polymorphic typed lambda calculus .
Technical Report CMU-CS-88-125, Carnegie Mellon University Computer Science Department, March, 1988.

[Reynolds and Plotkin 90]
Reynolds, J.C., and G.D. Plotkin.
On Functors Expressible in the Polymorphic Typed Lambda Calculus.
Technical Report CMU-CS-90-147, School of Computer Science, Carnegie Mellon University, July, 1990.

[Roscoe and Walker 88]
Roscoe, A.W. and E.G.L. Walker.
An Operational Semantics for CSP.
Theoretical Computer Science , 1988.
Submitted for publication.

6. RESEARCH IN UNIFORM WORKSTATION INTERFACES

User interface software is difficult and expensive to implement [Myers 89a]. Highly-interactive interfaces are among the hardest to create, since they must handle at least two asynchronous input devices (e.g., a mouse and keyboard); real-time feedback; multiple windows; and elaborate, dynamic graphics. Most graphical interfaces today are created using toolkits, which are collections of interaction techniques (sometimes called "widgets" or "gadgets"), such as menus, scroll bars, and buttons. Unfortunately, these toolkits are often difficult to use, since they contain literally hundreds of procedures. In addition, the toolkits often do not help the programmer create the most important part of the application—the graphics that appear in the main application window. Furthermore, it is usually difficult or impossible to modify toolkit items or create new ones. Our research in Uniform Workstation Interfaces was originally embodied in the Dante project. A year of research yielded no substantial progress, and Dante was thus replaced by the Garnet project, which aims to create a set of tools that will help user interface designers create, modify, and maintain highly-interactive, graphical, direct manipulation interfaces.

6.1. Motivations and Related Work

Garnet's goals follow closely those of its predecessor, Dante, with the exception that Garnet is not Mach-specific (as was Dante), and is being developed for any workstation running any X11 system with Lisp.

The primary goals of the Garnet project are:

- Demonstrate that the use of constraints and interactors makes the construction of user interfaces and user interface toolkits easier, more modular, and more modifiable.
- Create a small set of interactor objects that cover a wide range of user interface styles of interaction.
- Demonstrate that it is possible to provide graphical, direct manipulation construction tools that allow significant parts of the user interface to be constructed without programming.

A common name for software that builds user interfaces is a "User Interface Management System" (UIMS), and there are many examples of these. Unfortunately, most of these programs are very limited and are unable to create the types of interfaces that users wanted, and hence and have not been widely used. Notable exceptions include Apollo's Dialogue and Apple's MacApp.

Influences on the Garnet project include interaction technique layout tools such as the two Macintosh programs: Prototyper from Smethers Barnes and the Exper User Interface Builder. Examples from research labs include Menulay and one from DEC SRC. These programs allow the user interface designer to place preprogrammed menus, scroll bars, and buttons in windows, and then typically allow the designer to type in the name of a procedure that should be executed when the interaction technique is executed. These tools do not allow aspects of the interaction techniques themselves to be edited, however. The Garnet construction tools are designed to be as easy to use as these programs when the user only wants to assemble interaction techniques, but also to be more functional when new ones are desired.

Another important influence on the Garnet project is Apple's MacApp application development environment. MacApp provides an object-oriented framework that helps build programs with the standard Macintosh user interface. MacApp handles many of the details of the user interface, but leaves a number of hard problems to the application developer. In particular, while MacApp handles the menus and scroll bars *around* a window, it does not help much with mouse or keyboard events *inside* the window. For example, the application is told about mouse button presses and movement, and is required to deal with all issues of selection and moving objects itself. Garnet's Graphical Editor Shell attempts to go further in this area by helping to deal with input events inside the application's window also.

Garnet builds on earlier work by Dr. Myers on the Peridot UIMS. Peridot is in many ways similar to Garnet, but there is no programming interface to any Peridot features; it is a stand-alone program. Garnet has been designed so that its parts can be used independently by programmers, as well as by the Garnet construction tools.

To facilitate creating interaction techniques and application-specific graphic objects, our strategy was to separate the graphics from the *interactive behaviors*, which are the ways the graphics change when the user operates the input devices. In Garnet, many of the relationships among the graphic objects can be defined using *constraints*, which are declared once and then maintained automatically by the system. Like other interface builders, the Garnet interface builder allows existing toolkit items to be positioned, but it also allows new interaction techniques and application-specific objects to be created.

6.2. Constraints and Interactors

One goal of most user interface development environments and toolkits is to free the application from details of specific interactors and input device behaviors. This has proven to be a very difficult goal, and most previous attempts to achieve this separation have failed. Garnet has taken a new approach, and identified a few low-level input device behaviors. These are encapsulated into objects called *interactors*. There are a small number of different types of interactors, and each one handles a different kind of input device interactive behavior. For example, there are interactor types for menu behavior (selecting one from a set of objects), moving behavior (moving an object with the mouse), and angular rotation behavior (for interacting with circular gauges, etc.). In each case, the output graphics is entirely independent of the behavior, which allows tremendous flexibility.

A *constraint* is a relationship among graphical objects that is maintained even if one of the objects changes. For example, in an editor that supports boxes attached by arrows, the user interface designer can specify a constraint that the arrows must stay attached to the boxes. Then the boxes can be moved by the program or the mouse, and the lines will stay attached without any additional coding.

6.2.1. Constraints

An early version of constraints in Garnet was Coral [Szekely and Myers 88], although we now implement constraints in the KR object system. KR provides a prototype -instance model for objects, rather than the conventional class-instance model used by Smalltalk and **C**$^{++}$. In a prototype-instance model, there is no distinction between instances and classes; any instance can serve as a ‘‘prototype’’ for other instances. The advantages of the prototype-instance model are that it is much more dynamic and flexible than the familiar class-instance model. A high-level tool, such as the Garnet interface builder, can display a prototype on the screen, and allow the user to edit it. These edits are then automatically reflected in all instances of that prototype. For example, the designer might be changing the standard look-and-feel of the menu prototype, and immediately all menus in the system will change accordingly. In a class-instance model, it is much more difficult to change the class structure and have that reflected in instances [Giuse 89a].

Constraints in Garnet are arbitrary CommonLisp expressions stored in slots. When a program accesses a slot, it cannot tell whether the slot contains a simple value like a number, or a constraint that calculates the value. In the latter case, whenever the referenced slot of the other object changes, the formula is reevaluated.

The implementation of constraint satisfaction is designed to be very efficient. Constraints in Garnet are ‘‘one-directional.’’ This means that there always must be at most one formula for any slot. The result of this is that there can never be a choice about how to solve a constraint, so no planning is necessary. Therefore, when an object changes, the system always knows immediately which other objects to change, and how to change them. Surprisingly, this restriction does not substantially limit the interfaces that can be created, because it is almost always possible to find a one-directional way to specify any group of constraints.

An interesting feature of the constraints in Garnet is that the object referenced in the constraint can be accessed indirectly through a variable. For example, a feedback object in a menu (such as a highlight or outline) might be constrained to the same size as whatever object it should appear over. A slot holds the current object that the feedback should appear over, and whenever this slot is changed, Garnet would automatically reevaluate the formulas that depend on the slot, thus causing the feedback object to move. It is this mechanism that allows the interactive behaviors (described in section 6.2.2) to be independent of the graphics. For example, a menu interactor simply sets a slot with the object that the mouse is over, and the constraints ensure that the graphics that handle feedback are changed appropriately.

Constraints can also be used to connect graphical objects to application-specific objects. For example, the value of a gauge displayed on the screen could be constrained to a temperature value in the application.

Although the constraints are designed to be used for graphical objects, they are implemented in a general-purpose manner, and can therefore be used by applications for defining relationships on their own data, if desired. Since they are efficiently implemented, applications may find it convenient to use constraints for maintaining data integrity and dependencies, for example.

Other important aspects of the use of constraints in Garnet are:

- There is an easy-to-use, object-oriented language, KR, for specifying constraints in a prototype-instance fashion.
- Constraints can be defined in the abstract and dynamically attached to different objects.
- Constraints can be defined on lists of objects.

6.2.2. Interactors

One of the most difficult tasks when creating highly-interactive user interfaces is managing the mouse, keyboard, and other input devices. Typically, window managers or user interface toolkits only provide a stream of mouse positions and key events and require that the programmers handle all interactions themselves. Garnet tries to provide significantly more help through the use of *interactors* which are encapsulations of input device behaviors. The observation that makes this feasible is that there are only a small number of different kinds of behavior that are used in user interfaces. For example, although the graphics can vary significantly and the specific mouse buttons used may change, most menus operate in the same manner. Another example is the way that objects move around when following the mouse. Interactors capture these common behaviors in a central place while still being highly customizable by application programs [Myers 89b, Myers 90a].

Another advantage of the use of interactors is that it helps to separate and modularize the user interface software. The graphics are defined using the object-oriented graphics package and constraints, and the interactive behaviors are programmed separately using interactors. The interactors are connected to the graphics using constraints. The graphics of a user interface provide the ‘‘look,’’ while the interactors (connected to the graphics via constraints) determine the ‘‘feel.’’ Since interactors are completely ‘‘look’’ independent, any ‘‘look’’ can be linked with any ‘‘feel.’’

Interactors also provide a level of window manager independence. The designer is freed from details of how events are queued and how exception conditions are presented. The object-oriented graphics package and the interactors provide a complete layer hiding the details of the window manager. This should allow applications to be easily ported to various window managers.

In designing the interactors, there were many trade-offs that had to be considered.

Our design attempted to balance flexibility and power with ease of use. The earlier Peridot system only had very simple interactors, and multiple interactors were needed for many common operations. For example, to have an outline box follow the mouse while a button is pressed and have an object jump to the final position when the mouse button is released required three interactors: one to control the visibility of the feedback object, one to have it track the mouse, and one to have the actual object jump to the final position. In Garnet, interactors are higher level so that this behavior is achieved with one interactor. A result of this is that individual interactors have more parameters (to control the various options for feedback), and there are a few more interactor types than in Peridot.

An important design goal, however, was to limit the number of different types of interactors provided by Garnet. There are only six types and these handle almost all interactive behaviors in user interfaces.

- **Menu-Interactor**: for choosing one or more from a set of items, or for a single, stand-alone button.
- **Move-Grow-Interactor**: to move or change the size of an object or one of a set of objects using the mouse. This interactor can be used for one-dimensional or two-dimensional scroll bars, horizontal and vertical gauges, and for moving or growing application objects in a graphics editor.
- **New-Point-Interactor**: to enter one, two or an arbitrary number of new points using the mouse, for example for creating new lines or rectangles in an editor.
- **Angle-Interactor**: to calculate the angle that the mouse moves around some point. It can be used for circular gauges or for ‘‘stirring motions’’ for rotating.
- **Trace-Interactor**: to get all of the points the mouse goes through between start and end events, as is needed for free-hand drawing.
- **Text-string-Interactor**: to input a small (optionally multi-line) string of text.

This seems to provide an appropriate balance between ease of use (using the defaults) and flexibility (writing procedures). In the event new procedures are needed, the object-oriented implementation of interactors in CommonLisp should render it easy to create new interactor types.

6.3. Graphical Object System

Opal, the *object-oriented graphics component* of Garnet, allows the higher layers of the software to be independent of the details of the particular window manager used. In particular, this layer supports ‘‘retained graphics,’’ which means that the objects know where they are displayed in a window and are able to redisplay themselves automatically if the window is uncovered, and they can move and erase themselves [Kosbie et al. 90].

Previous implementations of the prototype-instance model have not supported changing the structure of instances. Opal provides special graphical objects called ‘‘AggreGadgets’’ which are used to hold a collection of other objects (either primitives or other AggreGadgets). When an AggreGadget is used as a prototype, its instances contain copies of the entire collection of objects. This means that an instance is made of each of the components of the AggreGadget, as well as for the AggreGadget itself. Changes to the AggreGadget are immediately reflected in all instances, including when components are added or deleted from the prototype. In this case, the corresponding components are immediately added or deleted from all instances. Thus when objects are erased, AggreGadgets can limit the number of other objects that are redrawn, and thereby improve efficiency [Dannenberg et al. 90].

6.4. Other Developments

6.4.1. An interface builder

On top of the Garnet toolkit layer (KR, Opal, constraints, and interactors) are a number of tools to assist the user interface designer. The most important is the Lapidary interface builder [Myers et al. 89a]. Lapidary provides a graphical front end to most of the underlying Garnet toolkit features.

In particular, Lapidary allows the designer, who does not have to be a programmer, to draw pictures of application-specific graphical objects which will be created and maintained at run-time by the application. This includes the graphical entities that the end user will manipulate (such as the components of the picture), the feedback that shows which objects are selected (such as small boxes on the sides and corners of an object), and the dynamic feedback objects (such as hair-line boxes to show where an object is being dragged). The designer creates prototypes of the objects in Lapidary, and then the application program creates instances of these as needed.

Lapidary supports the construction and use of interaction techniques, such as menus, scroll bars, buttons and icons. Lapidary therefore supports both *using* a predefined library of widgets, and *defining* a new library with a unique ‘‘look and feel.’’ The runtime behavior of all these objects can be specified in a straightforward way using constraints and abstract descriptions of the interactive response to the input devices. Lapidary generalizes from the specific example pictures to allow the graphics and behaviors to be specified by demonstration.

Lapidary is designed to allow the user interface builder flexibility in specifying object behavior. Graphical constraints can be attached to objects using iconic menus. If an object should move with the mouse, it can be selected and declared a feedback object. Lapidary will automatically generalize the constraints on the feedback object so they refer to whatever graphical object the mouse is over. Also, if an object should change based on some user action, the designer can specify this *by demonstration*. First, one state is drawn, and then another state, and Lapidary will automatically construct the constraints to change the object between the two states.

6.4.2. A dialogue box creation system

Sometimes it is easier to list the contents of a dialogue box or menu, rather than draw it meticulously. Jade automatically creates an attractively laid out dialogue box or menu from a simple listing of its contents. In addition to being simple to use, the specification passed to Jade has the additional advantage of being look-and-feel independent. The textual specification of the contents also describes the kind of input required (e.g., choice of one of a set, a number in a range, etc.), and the particular look-and-feel to use (e.g., Macintosh-like, Garnet-standard, etc.). From this, Jade will choose the correct interaction techniques, which themselves are designed using Lapidary. In addition, the heuristic rules that determine the placement of various parts of the interface are specific to a particular look-and-feel. For example, the set of buttons that make a dialogue box go away (''OK,'' ''CANCEL'') will be at the right for a Macintosh-like dialogue box, and at the top for a Xerox-Star-like one. [Vander Zanden and Myers 90]

6.5. Results

Version 1.1 of the Garnet toolkit was released in the fall of 1989, to enthusiastic response from the user interface development community. A second version was released the following spring. Although Garnet has only been working for a short time, it has already demonstrated that it makes the creation of graphical, highly-interactive user interfaces significantly easier. It is one of the few systems that supports the creation and exploration of various looks-and-feels for user interfaces. The use of constraints and automatic refresh for graphical objects has proven to be very useful and sufficiently efficient to support the desired interfaces. The encapsulation of the interactive behaviors makes it much easier to have the objects respond to input devices. The Lapidary interface builder allows more of the user interface to be specified graphically and by demonstration than any other interface builder, and Jade is the most advanced look-and-feel-independent dialogue box creation system. Taken all together, these components make Garnet an exciting and innovative system that is extending the state of the art in user interface software, while still being useful for creating user interfaces today.

Over 250 companies and universities have requested licenses for Garnet, with 85 sites already licensed by the end of the contract period.

6.6. Bibliography

[Dannenberg et al. 90]
Dannenberg, R.B., B.A. Myers, D. Giuse, D.S. Kosbie.
Using Aggregates as Prototypes.
In *The Garnet Compendium: Collected Papers, 1989-1990*. School of Computer Science, Carnegie Mellon University, 1990.
Available as technical report CMU-CS-90-154.

[Giuse 88] Giuse, D.
Lisp as a rapid prototyping environment: the Chinese Tutor.
Lisp and Symbolic Computation--An International Journal .
Kluwer Academic Publishers, 1988.

[Giuse 89a] Giuse, D.
KR: Constraint-based knowledge representation.
Technical Report CMU-CS-89-142, School of Computer Science, Carnegie Mellon University, April, 1989.

[Giuse 89b] Giuse, D.
Efficient frame systems.
Lecture Notes in Artificial Intelligence, EPIA '89.
In Martins, J.P., and E.M. Morgado,
Springer-Verlag, 1989.

[Giuse 89c] Giuse, D.
Efficient knowledge representation systems.
The Knowledge Engineering Review 4(4), 1989.

[Giuse and Baumeister 88]
Giuse, D. and L. Baumeister.
The Meta-Device: an object based, non-retained graphical layer .
Technical Report CMU-CS-88-136, Carnegie Mellon University Computer Science Department, April, 1988.

[Kosbie et al. 90] Kosbie, D.S., B. Vander Zanden, B.A. Myers, D. Giuse.
Automatic Graphical Output Management.
In *The Garnet Compendium: Collected Papers, 1989-1990*. School of Computer Science, Carnegie Mellon University, 1990.
Available as technical report CMU-CS-90-154.

[Kuokka and Giuse 88]
Kuokka, D. and D. Giuse.
The Dante application interface.
In *Proceedings of the 2nd Conference on Computer Workstations*. IEEE, March, 1988.

[Lane 90] Lane, T.G.
User interface software structures.
PhD thesis, School of Computer Science, Carnegie Mellon University, May, 1990.
Also available as technical report CMU-CS-90-101.

[Myers 88a] Myers, B.A.
The Garnet user interface development environment: A proposal.
Technical Report CMU-CS-88-153, Carnegie Mellon University Computer Science Department, September, 1988.

[Myers 88b] Myers, B.
The state of the art in visual programming and program visualization.
In *Graphics Tools for Software Engineering: Visual Programming and Program Visualization* . British Computer Society, March, 1988.
Also available as Carnegie Mellon technical report CMU-CS-88-114.

[Myers 88c] Myers, B.A.
A taxonomy of window manager user interfaces.
IEEE Computer Graphics and Applications 8(5):65-84, 1988.

[Myers 88d] Myers, B.
Tools for creating user interfaces: an introduction and survey.
Technical Report CMU-CS-88-107, Carnegie Mellon University Computer Science Department, January, 1988.

[Myers 89a] Myers, B.A.
User-interface tools: Introduction and survey.
IEEE Software 6(1):15-23, 1989.

[Myers 89b] Myers, B.A.
Encapsulating interactive behaviors.
In *CHI'89 Proceedings*. ACM, May, 1989.

[Myers 89c] Myers, B.A.
AI in demonstrational user interfaces.
In *A New Generation of Intelligent Interfaces: IJCAI-89*. , August, 1989.
Position paper for the IJCAI-89 Advanced Interface Workshop. No abstract appeared with the paper.

[Myers 90a] Myers, B.A.
A New Model for Handling Input.
ACM Transactions on Information Systems 8(3):289-320, July, 1990.

[Myers 90b] Myers, B.A.
Making it easy to create highly-interactive, graphical applications in Lisp.
XNextEvent: The Official Newsletter of XUG, the X Users Group 3(1), 1990.

[Myers 90c] Myers, B.A.
An object-oriented, constraint-based, user interface development environment for X in Common Lisp.
In *Proceedings of the Fourth Annual X Technical Conference*. January, 1990.

[Myers et al. 88] Myers, B.A., A. Schulert, S. Wallace, O. Densmore, and D. Goldsmith.
User interface toolkits: Present and future.
In *SIGGRAPH '88 Panel Proceedings*. SIGGRAPH, August, 1988.

[Myers et al. 89a] Myers, B.A., B. Vander Zanden, and R.B. Dannenberg.
Creating graphical interactive application objects by demonstration.
In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*. ACM, November, 1989.

[Myers et al. 89b] Myers, B.A., D. Giuse, R.B. Dannenberg, B. Vander Zanden, D. Kosbie, P. Marchal, E. Pervin, and J.A. Kolojejchick.
The Garnet Toolkit Reference Manuals: Support for highly-interactive, graphical user interfaces in Lisp.
Technical Report CMU-CS-89-196, School of Computer Science, Carnegie Mellon University, November, 1989.

[Myers et al. 90] Myers, B.A., D. Giuse, R.B. Dannenberg, B. Vander Zanden, D. Kosbie, P. Marchal, E. Pervin, A. Mickish, and J.A. Kolojejchick.
The Garnet Toolkit Reference Manuals: support for highly-interactive, graphical user interfaces in Lisp.
Technical Report CMU-CS-90-117, School of Computer Science, Carnegie Mellon University, March, 1990.
This is a revision of technical report CMU-CS-89-196 from November, 1989.

[Rubine and Dannenberg 87]
Rubine, D. and R. Dannenberg.
ARCTIC: Programmer's manual and tutorial.
Technical Report CMU-CS-87-110, Carnegie Mellon University Computer Science Department, June, 1987.

[Serra et al. 88] Serra, M., D. Rubine, and R. Dannenberg.
A comprehensive study of analysis and synthesis of tones by spectral interpolation.
Technical Report CMU-CS-88-146, Carnegie Mellon University Computer Science Department, June, 1988.

[Shaw et al. 89] Shaw, M., D. Giuse, and R. Reddy.
What a software engineer needs to know: I. Program vocabulary.
Technical Report CMU-CS-89-180, School of Computer Science, Carnegie Mellon University, August, 1989.

[Szekely 88] Szekely, P.
Separating the user interface from the functionality of application programs.
PhD thesis, Carnegie Mellon University Computer Science Department, January, 1988.

[Szekely and Myers 88]
Szekely, P. and B. Myers.
A user interface toolkit based on graphical objects and constraints.
In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, September, 1988.

[Vander Zanden 89]
Vander Zanden, B.T.
Constraint grammars - a new model for specifying graphical applications.
In *Human Factors in Computing Systems: Proceedings, SIGCHI '89*. Association for Computing Machinery, May, 1989.

[Vander Zanden and Myers 90]
Vander Zanden, B., and B.A. Myers.
Automatic, look-and-feel independent dialog creation for graphical user interfaces.
In *Human factors in computing systems: Proceedings SIGCHI '90*. Association for Computing Machinery, 1990.

[Wiecha et al. 89] Wiecha, C., S. Boies, M. Green, S. Hudson, and B. Myers.
Direct manipulation or programming: How should we design interfaces?
In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*. November, 1989.
Panel presentation chaired by C. Wiecha.

7. RESEARCH IN VLSI

The task of designing VLSI systems grows continually more difficult, pushed by increasing circuit density and wafer scale integration, and pulled by ever more ambitious application requirements. At the same time that designs grow more complex, the trend toward special purpose systems calls for a larger number of designs to be carried out. As a result, current design tools and methodologies are unable to keep pace. Carnegie Mellon's research in VLSI is aimed at designing tools and architectures themselves in such a way that the process of implementing new systems becomes more rapid and reliable.

Our VLSI research program includes the following tasks:

- Develop and distribute VLSI logic design validation tools that combine advanced functionality with the efficiency needed to make early validation of large designs feasible.
- Develop and prototype application-specific architectures that exploit technology and parallelism for large gains in cost and performance, along with methodological, software and hardware support for their design and deployment.

7.1. Special Purpose Architectures

7.1.1. SLAP

The SLAP (Scan-Line Array Processor) project is developing a highly parallel (100-1000 processor) SIMD linear array architecture for image computation and related applications [Fisher et al. 87a]. Our early work concentrated on hardware implementation and on the development of programming paradigms and tools.

We planned a three board system to fit in a Sun-3 cabinet: two array boards and one controller board. The controller board was based on a commercial microprocessor controlling an array instruction issue unit and several fast data transfer and storage units. Several SLAP chips mounted on the controller board facilitated connecting the array into a ring, and made a one board system yielding hundreds of MIPS possible. We anticipated that when fully populated with 128 SLAP chips, the system should execute some four billion sixteen bit operations per second.

At the same time, we were developing programming support, including a compiler, assembler and simulators for the entire system [Fisher and Highnam 88a]. One interesting aspect of the compiler work involved the expression of inter-PE communication in a functional style that allowed automatic scheduling and optimization of the interaction of communication and computation.

Concurrently, we brought up two compilers. One translates SLANG, our image-oriented high-level data parallel language, into assembly code. The other translates APPLY, the image-processing language developed by the Warp group at CMU, into

SLANG code that runs on a SLAP of any size. We will thus be able to run many programs developed for Warp.

Although the four-processor SLAP chip worked at speed on first silicon in both p-well and n-well processes, yield was low and clocking at full speed was tricky. We tuned the chip design for enhanced yield and timing margins, at the same time improving our testing capability. We fabricated and tested an improved version of the chip.

The other focus of effort was on the system interface/controller and its software support. Low-level tools for generating and linking array code and controller code were written and tested, and await final updating to reflect late changes in the controller design. Another software task was to update the code-generation interface of the SLANG high-level language compiler.

By the end of the contract period, we completed the benchmarking and analysis of our optimizing compiler for SLANG. The results showed that the compiler meets our design goals of producing nearly hand-quality code on image processing applications. Further, the results demonstrate an important complementarity between loop unrolling and constant propagation in these programs, and also demonstrate that our technique of directional analysis provides on the order of a 20% improvement in runtime for these applications. The compiler's effectiveness also stems from the extensive use of other standard optimizations, such as constant subexpression elimination.

Parallel graphics

As part of our SLAP applications work, we also studied the performance of a polygon rendering algorithm that scales gracefully between processor-per-pixel and processor-per-polygon approaches, and which can be implemented using incremental arithmetic. Our preliminary results show that some point in between the extremes usually gives the best efficiency for a given number of processing elements, and that MIMD implementations are typically twice as fast as SIMD implementations due to better load balancing.

7.1.2. Chess

Because of the enormous search spaces involved in chess, chess-playing machines provide the ideal environment to conduct parallel search experiments. ChipTest, the SUN-based system built around one of our move generator chips, was crowned the new ACM North American Computer Chess Champion in the fall of 1987. A new 2-processor chess machine, Deep Thought, was then built, with anticipated performance being around 2,000,000 nodes/sec. This represents about a factor of 4 increase in raw speed over the retiring champion. Because of algorithm improvement, the actual speed increase should be around a factor of 5. The new design was a single VME triple height, full depth board that plugs directly into a SUN workstation. Based on test results between ChipTest and Hitech (another Carnegie Mellon chess machine), we expected that once the chess knowledge in Hitech is merged with the new machine, a computer grandmaster would become a reality.

We planned to use simulated annealing to automatically tune the evaluation function parameters instead of acquiring the equivalent parameters from Hitech. The equivalent evaluation parameters were not always present in Hitech, and there were practical difficulties in extracting the parameters from Hitech.

The speed of the new machine was roughly equivalent to the maximum speed that Cray Blitz (the 2nd place finisher in the '87 ACM Computer Chess Championship) could ever attain running on 20 4-Processor Cray XMP/48s, assuming linear speedup for the 80 Cray XMP processors. ChipTest, the predecessor to the new machine, finished first in the same ACM event. The new machine was to serve as a prototype machine for running parallel search experiments, with each of it's two processors containing a single-chip move generator implemented in 3-micron CMOS, as well as a set of Xylink logic cell arrays to implement the evaluation hardware.

Gary Kasparov, the world chess champion, convincingly defeated Deep Thought, the world computer chess champion in a two game exhibition match held on Sunday October 22, 1989, although Deep Thought was running on the fastest hardware yet. Using 6 custom processors running on 3 Sun4-330 workstations, Deep Thought was searching over 1.6 million positions/sec. Kasparov nevertheless defeated Deep Thought in both games, proving that there is still a considerable gap between the best chess playing computer and the best human chess player [Hsu 90].

7.1.3. A coprocessor design environment

The coprocessor design environment project focused on developing a suite of hardware and software tools aimed at assisting the process of designing and deploying custom coprocessors within an existing application environment. The tools provide early feedback on eventual system performance as well as assistance in hardware and software interfacing.

We completed the logic design and layout of an MC68020-compatible coprocessor design frame, and designed an example coprocessor with raster graphics and data structure applications that exercise the most commonly used features of the frame [Chatterjee and Fisher 87a, Chatterjee and Fisher 87b].

Part of our work involved designing a simple language for specifying the instruction set of a coprocessor, along with the translations needed both to produce object code invoking the coprocessor and to emulate the coprocessor in response to a ''coprocessor missing'' exception. We have implemented an interface compiler that translates such a specification into a phase to be inserted in a C compilation that inserts coprocessor instructions, along with emulation routines to be used in performance prediction.

The implementation of our initial set of software tools for coprocessor interfacing led to testing an example coprocessor chip. Developments in the microprocessor market since the inception of this project made the 68020 less attractive as a host CPU, so we dropped our plans to refine the design environment for release. Instead, we are retargeted our efforts to use what we learned to study the architectural and programming issues raised by the use of programmable accelerators. We implemented a simple distributed event simulator to let us take some measurements on fine-grain parallel decompositions of inner loops, and used these results to guide further research on programming tools and hardware support.

7.1.4. Parallel programming

We worked on compiling data-parallel languages for shared-memory machines. Our long term goal was to promote portable parallel programming by developing an expressive and high-level programming model that could be efficiently implemented on a variety of architectures. Data parallelism appears to be an appropriate level of abstraction that allows the programmer to think in terms of the logical structure of the program, avoid overconstraining the algorithm, and postpone (or even ignore) the low-level details of scheduling, synchronization and load balancing. However, implementations of this programming paradigm have been largely on SIMD machines, and attempts to port this model to MIMD machines have been mostly limited to SIMD emulation, which has high overheads associated with startup and synchronization, and is not competitive with hand-coded solutions written in a less portable style. We suspected that the regularities in the data-parallel paradigm allowed for the application of advanced compilation techniques and extensive compile-time analysis to produce efficient code for various MIMD machines.

Based on the these ideas, we implemented a compiler for an experimental data parallel language called VCODE. The compiler was targeted for the Encore Multimax, a shared-memory machine. Initial experiments indicated that it is possible for an optimizing compiler to obtain 70-90% efficiency. The compiler generates C Threads code, which is then compiled by the native C compiler to produce final object code. This allows some flexibility in the choice of the target machine. We also implemented some of the runtime mechanisms on the CRAY Y-MP, and their speed compared well with the speed of the standard libraries on that machine.

The first version of a compiler that translates abstract data parallel code into efficient code for a shared memory multiprocessor was completed, but not all optimizations were in place. Nevertheless, the compiler was able to perform a detailed synchronization analysis that allowed a large amount of serial overhead to be eliminated.

We also completed the design of a simple and powerful data parallel extension to C. The extension adds just two features: a parallel loop with data renaming via index vectors, and an array partitioning construct that concisely expresses a large variety of data remappings. Unlike existing designs, these features provide dynamic data sizing and nested parallelism without adding automatic storage management or new datatypes.

7.2. Circuit simulation and verification

Many hardware systems can be viewed at some level of abstraction as communicating finite state machines. In analyzing a system of *N* processes, however, the number of states in the global state graph may grow exponentially with *N*. We call this problem the *state explosion problem*. Our approach to this problem is based on another observation about distributed programs. Although a given program may involve a large number of processes, it is usually possible to partition the processes into a small number of classes so that all of the processes in a given class are essentially identical. Thus, by devising techniques for automatically reasoning about systems with many identical processes, it may be possible to make significant progress on the general problem.

We have addressed the problem of devising an appropriate logic for reasoning about networks with many identical processes. *Indexed Temporal Logic* is a logic we developed based on computation trees. We make precise the idea that changing the number of processes in a family of identical processes should not affect the truth of a formula in our logic, by introducing a new notion of equivalence between networks of finite state processes. We prove that if two systems of processes correspond in this manner, a closed formula of our logic will be true in the initial state of one if and only if it is true in the initial state of the other. We have devised a procedure that can be used in practice to find a network with a small number of processes that is equivalent to a much larger network with many identical processes. We call this result the *collapsing theorem for networks with many identical processes*.

Our results indicate that it is possible to show that exactly the same formulas of our logic hold in a network with 1000 processes as hold in a network with two processes.

We also devised a methodology for verifying an n-bit random-access memories by simulating $O(n \log n)$ patterns. This technique was tested on a series of CMOS static RAM designs ranging from 4 to 4096 bits. As a benchmark, simulating the 114,689 patterns to verify the 4096-bit RAM required 30 hours of user CPU time on a MicroVax-II, exploiting 32-fold data parallelism. The elapsed time was over 3 weeks, however, due to serious thrashing by the virtual memory system. Furthermore, as the memory size grew, the simulation time grew roughly quadratically, since both the number of patterns and the time to simulate a single pattern grew. Each time the memory size increased by a factor of 4, the simulation time increased by 18-20x. The time could be decreased dramatically by mapping the simulation onto the Connection Machine.

In other work, we verified the register array portion of the SLAP data path. The array contained 32 registers of 20 bits each. Most interesting, it contained separate read and write ports that could operate on different (or the same) registers simultaneously. This required simulating $O(n \log^2 n)$ patterns to account for the potential interactions between reads and writes. This verification required 1.5 hours on a SUN-3/160. It demonstrated the utility of the methodology for a class of circuits (multi-ported memories) that is prone to design errors.

Symbolic simulations

Our work also addressed several fronts dealing with formal verification of hardware using a symbolic simulator. We made significant progress in developing systematic methodologies for verifying sequential circuits. Since combinational circuits already are straightforward to verify using a symbolic simulator, this was a very significant step forward [Beatty et al. 89].

For data-intensive circuits, the traditional approach of building global state graphs is unworkable. For example, a single 32-bit counter has billions of reachable states. We developed a variation of the temporal logic formalism described by others in our group that addresses this problem for synchronous machines. Instead of building the state graph explicitly, we used symbolic simulation and our efficient Boolean function manipulation routines to characterize both states in the global state graph and the machine's next state function symbolically. We showed how to verify such circuits under the assumption that input sequences are restricted to members of a regular language. We demonstrated the verification of circuits with some 10^{21} states in just a few minutes' CPU time.

Although we have been using symbolic simulation to verify circuits for some years now, our work fell short in terms of generality, ease of use, and degree of automation. We did not have a formal notation for specifying the desired circuit property, nor a method to generate simulation patterns directly from the specification. Instead, we derived symbolic simulation patterns by hand and argued informally that these patterns served to verify the desired properties. Furthermore, it was particularly cumbersome to verify operations requiring multiple state transitions, such as occurs in pipelined systems.

Our most recent work corrects this shortcoming by presenting a formal state transition model for digital systems using three-valued (0, 1, *X*) models, a formal syntax for expressing desired properties of the system, and an algorithm to decide whether or not the system obeys the specified property. Our specifications take the form of formulas mixing quantified Boolean expressions and elementary temporal logic operators. The class of properties that can be expressed with this notation is relatively restricted, as compared to other temporal logics. Nonetheless, we have found that we can readily express most aspects of synchronous digital systems. It is quite adequate for expressing many of the subtleties of system operation, including clocking conventions and pipelining.

We extended the COSMOS symbolic simulator to support this verification methodology. In addition, we wrote a preprocessor (in a dialect of Scheme), that gives powerful constructs for reasoning about vectors of circuit nodes and for constructing and manipulating Boolean expressions. The output of the preprocessor is a sequence of simulation commands for COSMOS.

Our experiments with this new system have been able to verify several styles of stack and RAM circuits. The performance has proved quite acceptable: less than 5 minutes of CPU time (on a DEC 3100) to verify memories of over 1K bits.

Verification of hardware controllers

The control portion of a complicated hardware system can usually be expressed as a finite state machine. Given such a finite state machine, one would like to assert and check properties about the sequencing of events within the system. Temporal logic is a formalism that is appropriate for specifying such properties. Previously, we have developed efficient methods for checking temporal logic specifications of finite state systems with at most a few thousand states. Unfortunately, a large hardware system with several devices acting in parallel may give rise to a gigantic state space, making a direct analysis intractable. Our work took three approaches to solving the state explosion problem.

In the first two approaches, we described a hardware system as a set of finite state concurrent processes that operate on shared Boolean state variables. The first method used a DAG representation for Boolean expressions to represent the state space of the system. By using this representation our verifier was able to find a bug in an asynchronous arbiter circuit with more than 60,000 states in under a minute. The entire state space could be represented using fewer than 300 nodes. We are currently testing the verifier on a number of other sequential circuits.

The second approach involved making use of symmetries in the circuit to produce a reduced automaton that correctly models the circuit behavior. Our goal was to verify properties of very large systems of concurrent processes, provided they possess certain known symmetry properties. We represented the symmetries of a system as a permutation group on its state variables. Under appropriate conditions, the system states could then be merged into equivalence classes corresponding to the orbits of the group. Unfortunately, in the general case, determining whether two states are in the same equivalence class was computationally intractable. Thus, our work focused on finding classes of permutation groups for which the reduction problem was tractable.

The third approach attempted to exploit the hierarchical structure of large hardware systems. Looking more closely at such systems, we often found that they were composed of a set of modules which interacted through well-defined interfaces. The individual modules were usually much simpler than the whole system and could be analyzed using the temporal logic verifier. Our goal was to be able to deduce properties of such a system from properties of its individual components. The basic idea involved the notion of a *communication protocol*. A protocol specifies the interaction between a group of modules. The protocols were supplied by the designers based on their knowledge of the system. By checking that each module implemented the protocol in an appropriate sense, it was possible to assert that any property of a component would also be a property of the entire system. Additional properties of the system could then be deduced from these component properties. Some hardware systems also contained many repetitions of a single type of component. Intuitively, it seemed possible to check some properties of a similar system with only a few repetitions of the component and deduce that the same properties hold of the original system. Analyzing the smaller system was often much easier than analyzing the original.

Cache coherency protocols

In past few years, many shared-memory multiprocessors have been developed to meet demands for increased performance and reliability. These machines usually have on the order of ten processors, each of which has a local cache to reduce memory bus contention. However, these designs do not scale well to larger numbers of processors. More recent efforts include complex multi-level caching schemes in an attempt to accommodate hundreds of processors. The largest hurdle in designing such machines is maintaining consistency between caches and main memory. There have been many proposed methods to maintain the desired consistency while still providing acceptable performance. Unfortunately, understanding exactly how these cache coherency protocols operate has proved difficult. In order to guarantee correct behavior, some method of formal verification is needed.

We considered two major classes of Cache coherency protocols: Those protocols based on a *snooping*, and those based on *linked lists*. Snooping is usually used for systems with a common memory bus. Each cache watches the common bus to coordinate its activity with the other caches. In a linked list protocol, each piece of data is associated with a linked list which gives the location of each copy of the data. To modify the piece of data, messages must be sent to each location. This method is common in message-passing systems.

We have been experimenting with the use of model checking techniques for the verification of such protocols. Since the protocols are designed to be implemented in hardware, they can be described by finite-state models. One of the protocols which we are currently attempting to verify is for an actual multiprocessor currently being developed (The Encore Gigamax). We view this work as a test of the practical applicability of our methods and as an opportunity to allow practical experience to drive the development of new methods.

7.2.1. COSMOS

COSMOS provides a combination of high simulation performance and a variety of simulation features. It simulates between 10 and 200 times faster than other switch-level simulators such as MOSSIM II. COSMOS achieves this performance by preprocessing the transistor network using a symbolic Boolean analyzer, converting the Boolean description into procedures describing the behavior of subnetworks plus data structures describing their interconnections, and then compiling this code into an executable simulation program [Bryant 87a].

An earlier bottleneck caused by the long time required to preprocess a circuit into an executable simulation program was solved by a combination of hierarchy extraction, incremental analysis, and assembly code generation. The preprocessor takes a flat network description and extracts a two-level hierarchy consisting of transistor subnetworks as leaves, and their interconnection as root. This extraction utilizes graph coloring/isomorphism-testing techniques similar to those used by wirelist comparison programs. To avoid repeating the processing of isomorphic subnetworks, it maintains a directory of subnetworks and their compiled code descriptions with file names derived from a hash signature of the transistor topology. Finally, the code generation program can generate assembly language declarations of the data structures rather than C code. The data structure formats for all Unix assemblers are sufficiently similar that the assembly code generator for a new machine type can be produced with minimal effort. As an example, a 1600 transistor circuit that earlier required 23 minutes to preprocess on a VAX-11/780 now requires only 2.9 minutes to preprocess the first time, and only 2.3 minutes subsequently.

Features of COSMOS include both logic and concurrent fault simulation, mechanisms to interface user-written C code to implement new simulation commands as well as behavioral models, and the ability to simulate up to 32 sets of data simultaneously. Programs are provided to translate circuit descriptions produced by the Berkeley Magic circuit extractor into the network format required by the symbolic analyzer.

Response from users has been favorable, especially regarding the simulation performance. We successfully simulated a number of chips containing over 40,000 transistors, with simulation speeds superior to any other switch-level simulator. For example, we were able to simulate a 4 processor SLAP chip (around 60,000 transistors) at a rate of less than 10 CPU seconds per clock cycle of simulation on a SUN-3/160. This enabled the designers to perform extensive simulation prior to fabrication.

We released an experimental version of COSMOS (Version 2.0) that is able to perform *symbolic* simulation. With symbolic simulation, the user provide simulation patterns containing Boolean variables, in addition to constants 0 and 1. The program then represents the states of the nodes as Boolean functions of the past and present input variables. By using OBDD's to represent the Boolean functions, we have been able to simulate many large and complex circuits symbolically. This style of simulation is made possible by the symbolic preprocessing performed by COSMOS which converts the transistor-level description of a circuit into a functionally-equivalent Boolean representation.

Symbolic simulation provides a convenient tool for formal circuit verification. We can determine the behavior of the circuit over far more combinations of inputs than would be conceivable by conventional simulation. For example, we recently used the symbolic simulator to verify an 80-bit priority encoder that is to appear in a new move generator for a chess machine. This circuit has 88 inputs, and hence exhaustive simulation would require on the order of 10^{19} years to simulate conventionally. (For reference, the big bang is believed to have occurred about 10^{10} years ago.) Using symbolic simulation, we were able to evaluate this circuit for all possible inputs and to verify the output values with 1.5 hours of CPU time on a MicroVax-II.

We studied the resource requirements of the COSMOS simulator using a 43,000 transistor benchmark circuit supplied by Intel. Running on a VAX 8800, COSMOS requires 30 minutes to create an executable simulator, which then simulates 1 clock cycle every 2.2 seconds. In contrast, MOSSIM II requires only 12 minutes to create its simulation data structures, but 23 seconds to simulate 1 clock cycle. Thus, COSMOS outperforms MOSSIM II for any simulation run longer than 16 cycles.

Our symbolic analyzer ANAMOS requires 13.2 MB of virtual memory to preprocess the circuit. Extrapolating to larger circuits, the virtual memory requirement will impose a circuit size limit of around 200,000 transistors. However, with a better partitioning of the preprocessing programs and with more careful coding, we should be able to decrease the memory requirements dramatically. Developing a simulator capable of handling million transistor circuits seems well within our reach.

ANAMOS II

We began a new version of ANAMOS, which had been in operation since 1986. One goal of our rewrite was to produce more optimized Boolean formulas. The original version of the program missed significant optimization possibilities by considering only small units of the circuit at a time. Consequently, the resulting symbolic description of a unit contained terms to handle conditions which would never arise during actual circuit operation. For example, when a signal and its complement were inputs to a unit, ANAMOS would treat these signals as if they were completely independent. As a result, the symbolic description contained terms of importance only when both signals are 1, or both are 0.

To find more optimizations, the new ANAMOS carries out the symbolic analysis on larger blocks of logic. In order to accomplish this, it is imperative that powerful Boolean function manipulation routines are employed. An experiment of using a Binary Decision Diagrams (BDDs) is currently in progress. Some preliminary results from this have been quite promising. For example, the symbolic description of a barrel shifter circuit reduces in size by an order of magnitude.

Today, the ANAMOS program takes a transistor network and performs a symbolic analysis using pairs of Boolean formulas to derive a behavioral description of the transistor circuit. Unfortunately, the program has some major deficiencies: it is very tightly coupled to the Cosmos switch-level simulator, it requires an excessive amount of memory, and it generates unnecessary large Boolean functions for some types of networks. The new ‘‘ANAMOS II’’ tries to rectify these problems, but also to make ANAMOS into a more general tool. During discussions with industry, it has become very clear that there is a need for such a tool that can take a transistor network and transform it into a multi-level ‘‘gate’’ network with the same behavior. This gate network can then be simulated/analyzed/... using standard simulators and hardware accelerators.

Delay modeling

We developed a transformation technique that allows comprehensive delay and race modeling in a symbolic simulator. Previously, only unit delay and zero delay modeling could be used for a symbolic simulator. Traditional race analysis algorithms are highly data dependent, and statements like ‘‘if the current value is 0 and is trying to change to 1 then ...’’ are very common. Clearly such algorithms cannot be used directly in a symbolic simulator where the current value of a node might be the Boolean formula $a+b$ and it is trying to change to $(a+b)c$. By exploiting specific properties of the dual-rail encoding used by COSMOS, we have adapted a ternary bounded-delay algorithm to transform a circuit into a ‘‘delay circuit.’’ This delay circuit can then be simulated by using a unit delay algorithm. However, the results of such simulation correspond exactly to the results that would be obtained by simulating the original circuit using the bounded delay algorithm. The most surprising part of this transformation, except for the fact that it actually can be done, is that the overhead is relatively small. For example, using a pure unit delay model it takes approximately 10 seconds of CPU time to verify the addition function for a 16-bit precharged ALU circuit. The same verification, with the delays in the nodes assumed to be bounded between 1.7ns and 4.5ns, takes about 33 seconds.

Test generation

We have extended the COSMOS symbolic simulator to generate tests for sequential MOS circuits by *symbolic fault simulation*. To generate tests for a circuit, the program simulates the behaviors of the good and faulty circuits over a sequence of input patterns containing Boolean variables. It creates representations of Boolean functions in terms of the variables describing the values on the primary outputs. It then derives a set of test sequences by determining assignments to the variables that will cause the good and faulty circuits to produce different outputs.

This approach to test generation has several important advantages over more traditional methods based on combinatorial search. It can generate tests for MOS circuits represented at the switch-level. It can generate tests for sequential circuits, where the input patterns contain sequences of Boolean variables to denote a multiple cycle test. Finally, the simulator-based user interface naturally allows the user to provide an indication of an overall test strategy, relegating the tedious task of detecting a specific set of faults to the program. That is, by simulating patterns with control inputs set to constants and data inputs set to variables, the user can indicate the means by which data are transferred from the primary inputs to the inputs to a subsystem, and from the subsystem outputs to the primary outputs.

We have successfully generated tests for sequential circuits of up to 700 transistors. This is over twice the largest previous switch-level test generation benchmark, and the first sequential one. Unfortunately, the virtual memory requirement grows rapidly for larger circuits. We believe that by tuning the code, the test generator can be made practical for true VLSI circuits.

Data parallel simulation

Most attempts to exploit parallelism in simulation utilize *circuit parallelism*. In this mode, the simulator extracts as much parallelism as it can while modeling the behavior of the circuit over a single test sequence. Such an approach has the advantage that it accelerates the existing way in which simulators are used. However, its performance is limited by the degree of parallelism found within the circuit, as well as the need to maintain synchronization. Furthermore, the overhead of keeping the simulation synchronized and communicating values between processing elements can overwhelm the savings gained through parallel evaluation.

We implemented a data parallel version of COSMOS on two different types of the machines. Unlike other switch-level simulators, COSMOS can evaluate the behavior of a circuit obliviously, since the new state of a set of nodes is computed by simply evaluating a sequence of Boolean formulas. In reality, our implementation is not entirely oblivious, since it maintains an event list to indicate which regions of the circuit should be evaluated. However, we queue a logic element on the event list if one of its inputs has changed for any of the test cases being evaluated. This leads to some unnecessary evaluation, but our experiments indicate that this does not compromise the performance significantly.

One implementation operates on conventional hardware, exploiting the bit-level parallelism inherent in machine-level logic operations (typically 32-fold). A set of library routines assist the user in writing C code to generate a number of test cases, evaluate them in batches of 32 at a time, and then check the simulation results. This typically yields a speedup of 20x over sequential simulation. It falls short of the optimal factor of 32 due to the overhead of packing and unpacking the data, and the decreased locality of simulation activity.

The second implementation runs on a Connection Machine. In this implementation, each processor stores the state of every circuit node for a single test sequence. The host executes the simulation program, commanding the processors to perform logic operations on their node values or to test whether a node has changed state. We were able to implement this version by modifying the existing simulation program to make calls to subroutines in the C/Paris library. This version achieves very high speed, since it does not require any use of the communication network in the Connection Machine.

We have evaluated the performance of data parallel simulation for a 16-bit nMOS ALU (688 transistors), based on a design in Mead and Conway. The ALU control inputs were set to perform addition. The patterns were generated automatically, and the results were tested by comparing the ALU output values to the input values. The Connection Machine was a Thinking Machines Corp. CM-2 with 32 physical processors, configured as 1M virtual processors, with a VAX 8800 as a host. The table below shows the time required to evaluate 32 million different input patterns.

Program	Machine	Parallelism	Absolute Time	Relative Time
MOSSIM II	VAX 11/780	1	2.53 yrs.*	376,000
COSMOS	VAX 11/780	1	81 das.*	33,000
COSMOS	VAX 11/780	32	103 hrs.*	1,743
COSMOS	SUN 4	32	10.7 hrs.*	181
COSMOS	CM-2	32K	212 secs.	1
Circuit	10 MHz	1	3.2 secs.*	1/66

* - Estimated by extrapolation

As can be seen, the Connection Machine achieves a very high speed-up over sequential evaluation. In contrast, for a simulator exploiting circuit parallelism, it is doubtful that many circuits contain 32,000-fold parallelism, nor that this degree of parallelism could be supported without high overhead in synchronization and communication.

We successfully mapped the simulation of a 4K CMOS static RAM (24,577 transistors) onto a 32K processor CM-2. This mapping exploits *data parallelism*, in which the behavior of the circuit over a number of independent patterns is evaluated simultaneously. We were then able to formally verify the RAM in 5.6 minutes by simulating 114,689 specially selected patterns. This was 250 times faster than performing the same simulation on a MicroVax-II.

Unfortunately, this method did not scale well to larger circuits. Preprocessing the circuit with ANAMOS requires 37 MB of virtual memory, since the entire memory array must be analyzed as a unit. Furthermore, our parallel mapping required 4 bits per node to store its state, plus extra space for storing temporary values. Thus, we were limited by the 64K bits of memory available to each CM-2 processor.

COSMOS and NECTAR

Despite the major advances in switch-level simulation performance over the past few years, we were still unable to perform full switch-level simulation of the largest chips being manufactured (more than 1 million transistors.) Our major limitation was not CPU speed so much as primary memory capacity. The data structures required to represent a large circuit were extremely large (hundreds of megabytes), and were accessed to heavily to make efficient use of virtual memory.

We believed the high performance distributed network system Nectar, also being developed at Carnegie Mellon, would be a good simulation engine for these simulations. By making the granularity coarse enough, we could ensure that the communication overhead was very small compared to the total simulation time. Whereas most experiments with parallel logic simulation produced disappointing results, we believed that it could be viable for handling circuits that are too large to run on a single machine.

Sequential circuits

We have developed a new method for specifying and verifying sequential circuits using the symbolic simulation capability of COSMOS. The heart of the method is to use a representation function to let a designer reason about an abstract version of his machine, rather than deal with the large and messy case analysis needed to deal directly with pipelined circuits. Given this approach, we have developed a way to use COSMOS not only to simulate the circuit, but also to evaluate the representation function and the verification conditions.

We developed a circuit verification approach with two novel features. First, it used Hoare-style representation functions to abstract the typically messy state of a (possibly pipelined) circuit, allowing the designer to write specifications as simple Hoare-style rules. The approach could then be extended to handle more complex styles of specification, but representation functions seem to allow a large number of difficult-looking problems to be easily specified. Second, it used the symbolic simulation capability of COSMOS not only to simulate every possible computation of the circuit, but also to evaluate the representation function and verification conditions. Using this approach, we verified that a synchronous systolic stack circuit of 96 cells and 5400 transistors works correctly, independent of circuit delays. This verification used about 4.5 minutes of CPU time and 31 MB on a VAX 8800.

A major goal of our research on circuit verification most recently has been the development of compositional and hierarchical reasoning techniques that are suitable for use with temporal logic model checking algorithms. Although there has been quite a lot of research on compositional techniques for reasoning about concurrent and distributed programs with manually constructed correctness proofs, there has been very little research on how this type of reasoning can be adapted to automata-based verification methods. We have investigated a method of compositional reasoning in which the environment of a module is modeled by another module called *interface module*, and modules are verified by checking their behavior in combination with the interface modules. This approach fits quite nicely with the verification methods that we have developed previously. Our work on analyzing synchronous circuits was based on a programming language called SML for describing complicated finite state machines. This language has many of the standard control structures found in modern imperative programming languages including nonrecursive procedures and processes. The SML compiler extracts a Moore machine from a high level description of the state machine as program, and the Moore machine can then be used as input to our model checking program or various CAD tools in the VLSI lab for generating layouts. For our research on compositional verification, we used our extension to SML, CSML, which allows us to describe complex hardware controllers in a hierarchical manner and to construct interface modules automatically. When we used this approach to reason about a CPU controller with decoupled access and execution units, we were able to reduce the number of states by a factor of 6.

A temporal logic based programming language

We developed a temporal logic based programming language for specification, simulation, and verification of digital circuits. The language is actually a general purpose programming language with temporal formulas as its Boolean expressions. The temporal operators are important for describing the time intervals used in specifying the behavior of synchronous circuits. The language includes both *future-time* operators and *past-time* operators. The past-time formulas are used for simulation, while the future-time formulas are used for verification. Most synchronous circuits of reasonable size can be simulated in a natural manner by programs in the language. The language can also be used for automatically verifying many important properties of such circuits.

7.2.2. Symbolic Boolean manipulation

In the process of verifying finite state machines, large Boolean expressions often occur. For this reason, the design of efficient algorithms for manipulating and testing Boolean expressions are important to the success of automatic verification methods. Most of the relevant tests on Boolean expressions can be reduced to a simple test for satisfiability, so much research has been devoted to this problem. The best theoretical upper bound for testing satisfiability is achieved by Allen Van Gelder's algorithm. He proves that his algorithm has a worst-case running time less than $2^{(0.25+\varepsilon)L}$, where L is the length of the Boolean expression to be tested. We have modified the algorithm to improve the worst-case running time, and we are currently developing a parallel implementation for the modified algorithm. Although the algorithm has a natural parallel decomposition, additional changes will be necessary in order to facilitate load balancing among the processors.

Our method for representing and manipulating Boolean functions as Ordered Boolean Decision Diagrams (OBDD's) has proved to be among the most efficient and reliable methods known [Cho and Bryant 89]. Several other organizations doing research on logic synthesis and verification (e.g., UC Berkeley, IBM, and Fujitsu) have implemented and are using our algorithms. We have implemented a new version of the algorithms that runs 10 to 40 times faster than the original implementation. It is now feasible to construct and represent the Boolean functions describing combinational logic gate networks with up to 3719 gates and 207 primary inputs.

We recently confirmed a conjecture that the Boolean functions representing the outputs of a multiplier provide difficult cases for the OBDD representation. That is, we proved that for an n-bit multiplier with outputs numbered from 0 (LSB) to $2n-1$, the Boolean function representing either output i or $2n-i-1$ requires an OBDD with at least 1.09^n vertices. Our experience has been that multipliers with word sizes greater than about 10 require graphs that are too large to handle efficiently.

In proving this lower bound, we discovered an interesting relation between the techniques used to prove lower bounds on OBDD sizes and to prove lower bounds on the area-time complexity of a VLSI implementation. In particular, the same form of proof that shows that any VLSI implementation of a single output Boolean function requires area-time complexity $AT^2=\Omega(n^2)$ also shows that any OBDD for the function must have c^n vertices for some $c>1$. In fact, our lower bound proof for multiplication is actually a lower bound proof for VLSI. It shows that computing output n of an n-bit multiplier has about the same area-time complexity as computing the entire product.

These theoretical investigations of Boolean function complexity have given us a great deal of insight into the strengths and weaknesses of different representations.

An alternate approach to BDDs

Another approach to binary decision diagrams that we explored is somewhat different from the one discussed above. We view the binary decision diagram for an n-argument Boolean function *f* as the minimal finite state machine for the set of Boolean vectors of length n that satisfy *f*. Because the minimal finite automaton for a regular language is unique up to isomorphism, it is easy to argue that this representation provides a canonical form for Boolean functions. Boolean operations involving NOT, AND, OR, etc., are implemented by the standard constructions for complement, intersection, and union of the finite languages accepted by these automata. In general, each of these operations involves building a *product* automaton and then minimizing it.

When we construct a binary decision graph, our algorithm follows the syntactic structure of the Boolean formula. First, the level of each Boolean operation is determined. Operations in the same level can be performed in parallel. If there are few operations at some level, then these operations are divided into a sequence of suboperations that can be processed in parallel.

7.2.3. Metastability

Synchronizer failures are among the most difficult hardware problems to diagnose because of their transient nature combined with their very wide range of mean times between failures (MTBF). This problem can be minimized by either using circuits that avoid synchronizers altogether (self-timed logic) or to carefully design synchronizers so that their MTBF is very high compared to other components.

In both cases, it is necessary to analyze the metastable behavior of the decision circuit subjected to asynchronous signals (usually some type of flip-flop). We have built a tester, using a MOSIS-fabricated printed circuit board, designed to measure the response time distribution for devices subjected to asynchronous input signals.

Despite the importance of reliable metastability data for circuit designers, this data is rarely published in common data books. This is partially due to the lack of standardized testing procedures. Our tester can be used to evaluate new device designs and to test off-the-shelf components.

The tester consists of two fast signal generators with a precise, programmable delay. The resolution of the programmable delay, about 20 ps, is fine enough to explore the region of metastability of most practical devices (TTL, CMOS, custom VLSI). The tester has two analyzers that can timestamp a signal transition with a resolution of about 1 ns.

7.2.4. Asynchronous circuits

Asynchronous systems have many possible advantages over traditional synchronous systems but are not widely used due to the unavailability of suitable components, a lack of experience with such systems, a lack of a simple, proven design method, and few tools to help the designer. The asynchronous systems project has designed a set of asynchronous building block parts and used them to design and build a number of small asynchronous systems for testing.

Work progressed on a system to automatically translate programs written in an Occam-like langauge into asynchronous circuits. Occam is a language similar to CSP used for describing concurrent communicating processes and turns out to be a very natural description language for a class of self-timed circuits. Programs are translated into initial circuits in a syntax directed way, and correctness-preserving optimizations will be done on the resulting circuit. The resulting circuit will be correct by construction in that the circuit will correctly implement the given program. Proving that the initial program is correct, however, remains a task for the user.

The result is a netlist of circuit modules and wires that can be simulated using switch-level simulators like COSMOS and RNL, or assembled into a VLSI chip using place and route software on a set of circuit elements implemented and tested previously by the asynchronous systems group.

Three test chips have been compiled by the system, assembled as VLSI chips by the MOSIS FUSION place-and-route service and fabricated. Two of the chips implement different versions of a two-to-one fifo-join module and are fully functional. The third implements a switch for cut-through packet routing similar to the Torus Routing Chip of Dally and Seitz. This chip consists of two two-way routers operating in parallel.

We built a second version of the chip implementing a switch for cut-through packet routing. The first version was only half functional (one of the two routers on each chip was fully functional) due to a problem with the interaction of the Magic layout program and FUSION. After that problem was corrected, a second version was fabricated and was mostly functional, this time the problem was a bug in the FUSION routing software. However, the second chip was functional enough (luckily, the problem is in the data path, not the control path) to verify that the generated circuit would have been correct if not for the unfortunate bug in the routing software used.

Component models in the verification of asynchronous circuits

The formal correctness of a circuit is relative both to a specification and to a component model. A component model is a formal description of the behaviors of the components from which a circuit is constructed. It is important to have a wide range of component models available. If a component model is too conservative (makes too few assumptions), then it may be impossible to verify the correctness of a circuit that works in practice. This reduces the applicability of formal verification. Worse yet, if a component model is too liberal (makes too many assumptions), then a circuit that is verified may not actually work in practice.

Historically, little attention has been given to the component models used in the verification of asynchronous circuits. Two primary component models have been used: The delay-insensitive model and the speed-independent model. However, neither deals with timing information like the speed with which circuit components respond to inputs.

Consequently, such models are too conservative to be used for a large class of circuits that actually work in practice.

We have expanded the class of component models that can be used by our automatic verification tools (both the trace theory verifier and the CTL model checker). It is now possible to verify circuits that rely on assumptions about the relative delays of their components for correct operation. This greatly expands the class of circuits that can be automatically verified and makes the verifier a more useful tool for the design of asynchronous circuits. The worst case complexity of the verification algorithm is linear in the size of the state graph for both the specification and the circuit. Bounded and unbounded maximum delays, bounded minimum delays, and setup times can all be handled with this technique. Moreover, the nondeterminism inherent in asynchronous circuits can be modeled more accurately by this technique than by other techniques that deal with timing.

We have demonstrated this method on an asynchronous queue. The circuit was known to contain an error, but our earlier work on this circuit could not be used to show that it would work correctly if this error was fixed, since the circuit made certain assumptions about the relative delays of its components. By using the debugging information provided by the verification algorithm described in the previous paragraph, we were able to design a correct version of the circuit and then demonstrate that it satisfied its specifications. This is the first time that such a circuit has been formally verified using its original timing assumptions. We plan to integrate our technique for modeling timing information with the technique described above for representing circuits using BDDs. We expect this to increase the size and complexity of circuits that can be efficiently verified.

CTL, trace theory, and timing models

We developed a system that combines CTL model checking and trace theory for verifying speed-independent asynchronous circuits. This system is able to verify a large and useful class of liveness and fairness properties, and is able to find safety violations after examining only a small fraction of the circuit's state space in many cases. An extension was implemented that allows the verification of circuits that are not speed-independent, but instead rely on assumptions about the delays of their components for correct operation. This greatly expands the class of circuits that can be automatically verified, making the verifier a more useful tool in the design of asynchronous circuits.

The system has been demonstrated on several fair mutual exclusion circuits, including a speed-independent version that is verified correct. It has also been used to show that given quite weak assumptions about the relative delays of components, the problem of designing a fair mutual exclusion circuit using a potentially unfair mutual exclusion element becomes almost trivial. Other examples verified with the system include a self-timed queue element.

7.2.5. Alternative state-space representations

Traditional temporal logic model checking techniques have been limited to control circuits, since they require a complete enumeration of the state space, and the number of states of the data path part of a circuit is prohibitively large. Recently, however, we have carried out experiments in which we modified the original temporal logic model checking algorithm to represent a state graph using binary decision diagrams. Because this representation captures some of the regularity in the state space of sequential circuits with data path logic, we have been able to verify circuits with an extremely large number of states. For example, using this method, we verified a synchronous pipeline with approximately 5×10^{20} states. Our model checking algorithm handles full CTL with fairness constraints. Thus, we are able to handle a number of important liveness and fairness properties, which would otherwise not be expressible in CTL. The empirical results we have obtained on the performance of the algorithm applied to both synchronous and asynchronous circuits with data path logic indicate that the approach is very promising. More work is required in this area, however, to characterize these algorithms and determine exactly when they provide improved performance over traditional methods. The same basic idea should be useful in a number of other algorithms that deal with large state transition graphs. For example, we are currently trying to adapt this technique to various algorithms for CCS equivalence.

The state explosion problem can be particularly severe in the case of asynchronous circuits and protocols. Even a minor mistake in such a system may cause events to become highly disordered and virtually impossible to analyze. In one asynchronous circuit that we considered, a relatively minor error resulted in a global state graph with more than 20,000 states. When the error was fixed the circuit had fewer than 200 states. A promising approach to the state explosion problem for this type of circuit is based on the use of partially-ordered computation models (which consider the causal ordering between events, rather than their temporal ordering). Since commuting events, which would normally determine different interleavings, correspond to the same partial order, the total number of cases that need to be considered in analyzing such a system may be significantly less. Model checking algorithms based on such models may have lower complexity than the algorithms we have developed previously. Such a result would be of substantial practical interest and would vindicate the theoreticians who have argued for partially ordered models of concurrency instead of interleaving models.

7.3. Bibliography

[Annaratone et al. 87]
Annaratone,M., F. Bitz, J. Deutch, L. Hamey, H.T. Kung, P. Maulik, H. Ribas, P. Tseng, and J. Webb.
Applications experience on warp.
In *Proceedings of the 1987 National Computer Conference*. AFIPS, 1987.

[Beatty and Bryant 88]
Beatty, D.L. and R.E. Bryant.
Fast incremental circuit analysis using extracted hierarchy.
In *Proceedings of the 25th Design Automation Conference*. ACM and IEEE, June, 1988.

[Beatty et al. 89] Beatty, D.L., R.E. Bryant, and C-J. Seger.
Synchronous circuit verification by symbolic simulation: An illustration.
In *Proceedings of the 6th MIT Conference on Advanced Research in VLSI*. MIT, October, 1989.

[Bose and Fisher 89a]
Bose, S., and A.L. Fisher.
Verifying pipelined hardware using symbolic logic simulation.
In *Proceedings of the International Conference on Computer Design*. IEEE, 1989.

[Bose and Fisher 89b]
Bose, S., and A.L. Fisher.
Automatic verification of synchronous circuits with temporal logic and symbolic logic simulation.
In *IFIP Workshop on Applied Formal Methods for Correct VLSI Design*. IFIP, 1989.

[Brunvand and Sproull 89]
Brunvand, E., and R. Sproull.
Translating concurrent programs into asynchronous circuits.
In *Proceedings of the International Conference on Computer Aided Design* . IEEE, 1989.

[Bryant 87a] Bryant, R.E.
Boolean analysis of MOS circuits.
IEEE Transactions on Computer-Aided Design of Integrated Circuits CAD-6(4):634-649, 1987.

[Bryant 87b] Bryant, R.E.
Algorithmic aspects of symbolic switch network analysis.
IEEE Transactions on Computer-Aided Design of Integrated Circuits CAD-6(4):618-633, July, 1987.

[Bryant 88] Bryant, R.E.
Data-parallel, switch-level simulation.
In *Proceedings of the IEEE International Conference on Computer Aided Design* . IEEE, November, 1988.

[Bryant 89] Bryant, R.E.
Formal verification of memory circuits by switch-level simulation.
Technical Report CMU-CS-89-156, School of Computer Science, Carnegie Mellon University, June, 1989.
Accepted for publication in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[Bryant 90] Bryant, R.E.
A methodology for hardware verification based on logic simulation.
Technical Report CMU-CS-90-122, School of Computer Science, Carnegie Mellon University, March, 1990.
Accepted for publication in the *Journal of the ACM*.

[Bryant 88] Bryant, R.E.
Verifying a static RAM design by logic simulation.
In *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI* . MIT, March, 1988.

[Bryant and Seger 90]
Bryant, R.E. and C.J. Seger.
Formal verification of digital circuits using symbolic ternary system models.
In *Proceedings of the Workshop on Computer-Aided Verification*. Rutgers University, June, 1990.
Also available as technical report CMU-CS-90-131.

[Chatterjee and Fisher 87a]
Chatterjee, S. and A.L. Fisher.
A coprocessor design environment.
In *International Workshop on Hardware Accelerators*. Oxford University, September, 1987.

[Chatterjee and Fisher 87b]
Chatterjee, S. and A.L. Fisher.
Low-end DMA coprocessor speedups: case studies.
In *IEEE International Conference on Computer Design*. IEEE, October, 1987.

[Cho and Bryant 89]
Cho, K. and R.E. Bryant.
Test pattern generation for sequential MOS circuits by symbolic fault simulation.
In *Proceedings of the 26th Design Automation Conference*. ACM and IEEE, June, 1989.
Also appeared in *Proceedings of TECHCON-88*.

[Clarke and Browne 87]
Clarke, E.M. and M.C. Browne.
SML—A high level language for the design and verification of finite state machines.
HDL Descriptions to Guaranteed Correct Circuit Designs.
In Borrione, D.,
North Holland, 1987.

[Clarke and Grumberg 87]
Clarke, E.M. and O. Grumberg.
Avoiding the state explosion problem in temporal logic model checking algorithms.
In *Proceedings of Sixth Annual ACM Symposium on Principles of Distributed Computing*. ACM, August, 1987.

[Clarke et al. 87] Clarke, E.M., M.C. Browne, and O. Grumberg.
Characterizing Kripke structures in temporal logic.
In *Proceedings of TAPSOFT/CAAP*. TAPSOFT/CAAP, April, 1987.
Also accepted for publication in a special issue of *Theoretical Computer Science*.

[Clarke et al. 89a] Clarke, E.M., D.E. Long, and K.L. McMillan.
A language for compositional specification and verification of finite state hardware controllers.
Technical Report CMU-CS-89-110, School of Computer Science, Carnegie Mellon University, January, 1989.

[Clarke et al. 89b] Clarke, E.M., D.E. Long, and K.L. McMillan.
Compositional model checking.
Technical Report CMU-CS-89-145, School of Computer Science, Carnegie Mellon University, April, 1989.

[Dill 88] Dill, D.L.
Trace theory for automatic heirarchical verification of speed-independent circuits.
PhD thesis, Carnegie Mellon University Computer Science Department, February, 1988.

[Fisher and Bryant 89]
Fisher, A.L. and R.E. Bryant.
Performance of COSMOS on the IFIP workshop benchmarks.
In *Proceedings of the IFIP Workshop on Applied Formal Methods for VLSI design*. IFIP, November, 1989.

[Fisher and Highnam 87]
Fisher, A.L. and P.T. Highnam.
Computing the Hough transform on a scan line array processor.
In *IEEE Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*. IEEE, October, 1987.

[Fisher and Highnam 88a]
Fisher, A.L. and P.T. Highnam.
Programming considerations in the design and use of a SIMD image computer.
In *Frontiers of Massively Parallel Computing*. IEEE, October, 1988.

[Fisher and Highnam 88b]
Fisher, A.L. and P.T. Highnam.
Communication and code optimization in SIMD programs.
In *Proceedings of the 1988 International Conference on Parallel Processing*. University of Pennsylvania, August, 1988.

[Fisher and Zsarnay 88]
Fisher, A.L. and J.A. Zsarnay.
System support for a VLSI SIMD image computer.
In *Proceedings of the IEEE Workshop on VLSI Signal Processing*. IEEE, November, 1988.

[Fisher et al. 87a] Fisher, A.L., P.T. Highnam, and T.E. Rockoff.
Architecture of a VLSI SIMD processing element.
In *IEEE International Conference on Computer Design*. IEEE, October, 1987.

[Fisher et al. 87b] Fisher, A.L., P.T. Highnam, and T.E. Rockoff.
Scan line array processors.
In *International Workshop on Hardware Accelerators*. Oxford University, September, 1987.

[Fisher et al. 87c] Fisher, A.L., P.T. Highnam and T.E. Rockoff.
A SIMD processing element.
In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE, October, 1987.

[Fisher et. al. 88] Fisher, A.L., P.T. Highnam, and T.E. Rockoff.
Scan line array processors: work in progress.
In *Proceedings of the Image Understanding Workshop*. DARPA, April, 1988.

[Hsu 90] Hsu, F.H.
Large scale parallelization of alpha-beta search: an algorithmic and architectural study with computer chess.
PhD thesis, School of Computer Science, Carnegie Mellon University, February, 1990.
Also available as technical report CMU-CS-90-108.

[Kravitz and Bryant 88]
Kravitz, S.A. and R.E. Bryant.
Massively parallel logic simulation.
In *Proceedings of TECHCON-88*. Semiconductor Research Corp., October, 1988.

[Kravitz et al. 89a] Kravitz, S.A., R.E. Bryant, and R.A. Rutenbar.
Massively parallel switch-level simulation — a feasibility study.
In *Proceedings of the 26th Design Automation Conference*. IEEE, June, 1989.
Submitted for publication in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[Kravitz et al. 89b] Kravitz, S.A., R.E. Bryant, and R.A. Rutenbar.
Logic simulation on massively parallel architectures.
In *Proceedings of the International Symposium on Computer Architecture*. May, 1989.

[Seger and Bryant 89]
Seger, C.-J. and R.E. Bryant.
Modeling of circuit delays in symbolic simulation.
In *Proceedings of the IFIP Workshop on Applied Formal Methods for VLSI Design*. IFIP, November, 1989.

GLOSSARY

3-DFORM	A general framework or system for representing 3-D models and relationships; it is based on the Framekit language
ALOE	A Language Oriented Editor
ALVINN	Autonomous Land Vehicle In a Neural Network; a connectionist network autonomous road-following system
ANAMOS	The VLSI project's symbolic analyzer
Apply	An image-processing language developed by the Warp group
ARL	Action Routine Language; a language for specifying semantics in Gandalf
AC	Aspect Change; the move from one topologically equivalent class in image recognition to another
Avalon	A set of high-level language primitives that allow easy access to Camelot, a distributed transaction facility
Camelot	A machine-independent, high-performance, distributed transaction facility
Cascade Correlation	A learning architecture that begins with a minimal network and adds hidden units one by one to eliminate remaining network errors
CCS	Calculus of Communicating Systems; a formal model of concurrent systems
Chip Test	A single-processor, Sun-based chess move generator system
CMSL	A specialized extension to SML that allows the programmer to describe complex hardware controllers hierarchically and to construct interface modules automatically
COSMOS	COmpiled Simulator for MOS circuits
Constraints	Relationships among graphical objects which are maintained when one of the objects is changed
CTL	A branching-time temporal logic used in much of the VLSI work in hardware verification
DA	Derivational Analogy; a learning method within Prodigy
Deep Thought	A two-processor chess machine developed by the VLSI project
Dichromatic Reflection Model	A theoretical reflection model that mathematically describes the physical cause of an object's highlights and color
EBL	Explanation-Based Learning module within Prodigy
ESS	Ergo Support System
Forsythe	An Algol-like language which emphasizes a close connection to the lambda calculus

Framekit	The language that is used with Vantage
Fusion	The MOSIS place-and-route service
Gandalf	A programming environment generator
Hitech	A search-intensive chess machine
Interactors	Objects in Garnet which control graphical device behaviors
Jade	Garnet's dialogue box creation system
Janus	A programming language used to specify views in Gandalf
KR	An object-oriented language used for specifying constraints in Garnet
Lapidary	A Garnet interface builder that allows flexibility in specifying object behavior
LC	Linear Shape Change; changing the orientation of an object while remaining in the same aspect classification
LCC	Local Consistency Check; one of two production phases explored in SPAM/PSM
LexGen	An editor that specifies scanning routines that validate user input in Gandalf
Nectar	A high-performance distributed network system that serves as a simulation engine for switch-level simulations of large chips
OBDD	Ordered Boolean Decision Diagram
Opal	The object-oriented graphics component of Garnet
OPS5	A production system computational model
ParaOPS5	A **C**-based optimized implementation of OPS5
Peridot	An early predecessor to Garnet
Photogram	A photogrammatic measurement module in SPAM
PRODIGY	A computational architecture designed as a general testbed for research in problem solving, planning, and machine learning
Quickprop	A new learning algorithm in which each weight takes a nearly optimal-sized step after each training cycle
RGB Space	Calculating illumination color and color consistency by explicit reference to the red/green/blue (RGB) components
RTAQ	Rapid Task AcQuisition
RTF	Region To Fragment; the second of two production phases explored in SPAM/PSM
RuleGEN	A compiler that converts knowledge represented as schemata into OPS5 productions
SCS	School of Computer Science
SLANG	An image-oriented high-level data parallel language

SLAP	Scan Line Array Processor
Soar	An architecture for general intelligence
SPAM	System for Photointerpretation of Airports using MAPS
SPAMEvaluate	An interactive tool for analyzing the results of a SPAM run
SPAM/PSM	A reimplementation of SPAM in ParaOPS5
SPATS	An automated performance analysis system for SPAM
STRIPS	A well-known AI planner out of Stanford
Strongbox	An integrated set of security tools provided by Camelot
TAQL	Task AcQuisition Language
TransformGen	An environment which automatically converts programs to run on a new grammar
UIMS	User Interface Management System; often used interchangeably with *UIDE*
UIDE	User Interface Development Environment; referred to in this report as *UIMS*
Vantage	The solid modeler that was specifically designed to support IU research and which is incorporated into the VAC
VAC	Vision Algorithm Compiler; a compiler that automatically generates vision recognition programs for a well-defined vision task
VCODE	An experimental data parallel language

References

Table of Contents