APL: A PROTOTYPING LANGUAGE

Robert Bernecky
I. P. Sharp Associates Limited
APL Systems Development Group
2 First Canadian Place, Suite 1900
Toronto, Ontario M5X 1E3
Canada
(416) 364-5361

ABSTRACT

The use of APL as a language for system design and prototyping is discussed. Benefits of APL over traditional design techniques are shown to include higher productivity, improved code reliability, superior maintainability and performance, and executable documentation.

Hierarchical and hybrid approaches to modelling systems of various degrees of complexity are presented, with examples chosen from the author's experience.

INTRODUCTION

The creation of any system is simplified if the system is understood before implementation begins. In the design of complex computer applications, such as interpreters, compilers, and session managers, executable models or prototypes of the application are an aid to understanding. They also provide other benefits, such as improved code reliability, higher programmer productivity, superior maintainability and performance, and executable documentation. The following presents the why and how of using APL for prototyping, with practical examples.

WHAT IS A PROTOTYPE?

A **prototype** or **model** is an executable program which simulates part or all of a system. Some major functions of prototypes are:

- to assist in the design of algorithms
- to assist in the design of data structures
- to estimate performance
- to validate system integration
- to serve as a debugging tool
- to help the user understand the system

EXECUTABLE PROTOTYPES

Non-executable techniques, such as "pseudocode" or flowcharts, are inferior to an executable model. The subjective "execution" of pseudocode by a human may mask serious design errors, whereas the execution of a working model by a computer will turn up errors which ever-optimistic programmers consistently miss. The importance of this objective, unbiased nature of computer program execution in analysis and design cannot be overemphasized. Consider an expression such as:

Paris in the
the springtime

Anyone who read the above as "Paris in the springtime" needs executable prototypes. This may seem obvious, but programmers and mathematicians exude a Panglossian attitude which persists in spite of working in environments where much of their time is spent in correcting the relation between reality and their conception of reality. This time is referred to as "debugging time" or "research" -- by programmers and mathematicians, respectively.

BENEFITS OF PROTOTYPING

Prototyping has a number of benefits over other design techniques. One of the most important occurs in system integration. It keeps you from getting that sinking feeling two-thirds of the way through a project, when it becomes obvious that two major pieces of it do not, and cannot, be made to fit together. If a prototype is built first, the pieces have already been integrated, and a potentially expensive redesign has been avoided.

Another advantage appears in the context of "doing it" versus "doing it right". In the course of constructing a large system, it usually becomes clear that certain algorithms, interfaces, or data structures could benefit from partial or total redesign [Wy13].

If these discoveries are made during the modelling process, it is often possible to perform the redesign immediately, with significantly less effort than during the actual implementation phase.

Performance measurement is simplified by use of prototypes. One of the virtues of an executable model, particularly a model which permits human interaction, is that it is easy to build instrumentation and monitoring functions into the model on a permanent or ad hoc basis. With proper measuring tools, it is possible to quickly answer questions such as:

- Which facilities are used most often?
- Which data structures are accessed most frequently? In what way?
- Will a linear search suffice, or is an index required?
- How does this data structure behave under different stimuli?
- How well will a given facility perform under light load? Under heavy load? Under overload?
- What sorts of errors are being made by the users?

Instrumentation may be analytical (producing tables, plots, histograms, counters), visual (displaying the parse tree generated by the syntax analyzer) or any other kind. The only rule is: If it helps the design process, use it. The use of a powerful, interactive language, such as APL, allows rapid ad hoc data collection and analysis, with minimal effort. For example, functions may be written to assist with topological analysis or to perform time computations during model execution.

Models provide excellent documentation for a system because they can combine the specifications, structure, and algorithms into a single, compact document. For this reason, proposed extensions to a system are easily analyzed at the model level.

Often, in the course of system maintenance, one encounters code which is apparently useless or incorrect. Having a model of the system available often helps to show why such code is either necessary or nugatory.

Models offer an opportunity to optimize at a high level, early in the design phase. A concise model often highlights areas where significant time or space improvements may be achieved, whereas the actual implementation may be complicated enough to hide these same areas.

WHY PROTOTYPE IN APL?

Prototypes may be written in any language, but the use of APL offers significant advantages over more primitive languages such as PASCAL or PL/I. APL possesses certain important characteristics which speed development and assist thought. Executability is important, but so are human interaction, powerful functions and operators, sub-second response time, and rapid creation and modification of functions.

Algorithmic analysis and design are assisted by the functions and operators in APL. Human minds can only deal with about 7 to 9 "chunks" of information at once, due to short-term memory limitations (I had the bibliographic reference for this in my head a minute ago, but forgot it while I was typing this line). The density of concepts embedded in an APL expression allows the mind to grasp at once a larger portion of the problem than is possible in other languages.

The functions and operators in APL are tools of thought. They hide the details and allow concentration on the problem at hand. The rapidity of development in APL allows study and comparison of alternate algorithms in a short period of time.

The interactive nature of the language speeds debugging efforts, even by such simple facilities as being able to display the value of an array in human readable form by entering its name, rather than by having to stare at a hexadecimal storage dump. The ability to generate test scripts with ease under program control makes vetting of the design faster.

The executability of the model speeds fault location in the actual implementation. If the model works, but the implementation fails, the error lies in the transcription of the model to the implementation, and the model may be used in conjunction with the implementation to isolate the fault. If the model also fails, a design error exists, and the model may be used by itself to isolate the fault interactively.

The rapid response time of APL systems (typically less than 0.1 second on most commercially available systems) increases human productivity and reduces frustration.

HIERARCHICAL PROTOTYPING

Just as the reasons for writing prototypes vary, depending on perceived needs, so do the approaches taken to writing them. A prototype designed to prove the feasibility of an algorithm may differ significantly in structure and detail from one written to analyze data structure performance.

The completeness and level of detail of a prototype will also vary depending on the problems to be solved by the prototype. A new, ill-understood project, such as a language compiler, will benefit from a relatively complete model. On the other hand, a small, well-specified problem, such as replacing the symbol table search functions for the same compiler, may be successfully modelled without requiring a model of the entire compiler.

One technique the author has used for creating prototypes is **hierarchical prototyping**, also called **levels of abstraction** [Di68]. This has worked well for simple, as well as complex systems. The basic idea is to begin by writing a model which helps the designer and the end user to understand the problem and its solution, without too much concern being given to actual details of implementation. Once this is completed, successively more detailed

models are written, which converge on the actual implementation, by mimicking actual code and data structures of the target system. In the author's experience, 3 levels of models have sufficed: Concise, intermediate, and detailed models, each with different objectives.

The Concise Model

The **concise model** or **conceptual model** has as its major objective the comprehension of the problem and proposed solutions to the problem. It serves as a basis for exploring ideas and various algorithms. Minimal consideration is given to the actual implementation, since this distracts from the important goal at this point, which is understanding the problem and its solution. The language used is pure APL. Anything goes, as long as it helps the designer's thought processes. This is not the time to be thinking much about details of representation (data structures). For example, a symbol table might be represented as a vector of enclosed arrays, and be searched by the **indexof** function, even though the final implementation might be very different.

The concise model is used to establish feasibility, and to develop algorithms. The conceptual freedom obtained from abstraction of data and program structures is invaluable at this point, as is the flexibility of that abstraction, if changes prove necessary.

For example, in the course of modelling the inner-product operator compiler for SHARP APL in 1978, the algorithms were redesigned several times in order to take advantage of new concepts which arose during the modelling process. Significant gains in performance resulted, and new applications became practical to solve in APL. The cost of applications such as cross-tabulations and transitive closure dropped dramatically, due to a thousand-fold speedup in the performance of Boolean inner products, such as $\alpha \vee . \wedge \omega$. Application of the same technique to more traditional inner products produced code which ran about twice as fast as FORTRAN.

The Intermediate Model

Once the problem and solution are understood, the task has been reduced to a programming exercise. At this point, modelling is used more as a programming aid and less as a tool of thought.

The major objectives of the **intermediate model** are to introduce the data structures of the actual implementation, and to create program structures which reflect those which will be used in the implementation. For example, a recursive function in the concise model might be replaced by an iterative function in the intermediate model, and the **indexof** function used to search a symbol table in the concise model might be replaced by a sophisticated binary tree search. The intermediate model may bear little or no physical resemblance to the concise model.

The intermediate model should actually work and produce results. It is prudent to write a driver function to exercise the intermediate model against the concise model, to ensure that they are in fact both solving the same problem.

The intermediate model is often the first concrete model of the entire system, but it is not unusual to have a model which mixes concise model and intermediate model components, for reasons of efficiency or convenience. For example, if a group of programmers are writing a compiler using a concise model as their paradigm, each may use the concise model as a driver for their components, replacing specific concise model functions with intermediate model equivalents. This allows each developer to proceed independently of the others.

Because validation of data structures and program structures is the goal, the actual APL code used in the model remains of lesser importance than verifying that the system can be built and will work as planned.

The Detailed Model

When the intermediate model is functional, work begins on a low-level, **detailed model**. This model should reflect, as directly as possible, the implementation language which will be the final product. The major objective of the detailed model is to make the production of implementation language code a transcription process. The detailed model is created by "primitivizing" the intermediate model. Primitivizing is the process of replacing APL constructs with functionally equivalent constructs which can be directly translated into known constructs in the implementation language. (Originally, the term "simplified" was used, but since the expressions resulting from the action were in fact usually more complicated, the term "primitivizing" was adopted.) A vector reversal might be primitivized in this way (All examples in this paper assume index origin 0):

```
z←⌽ω ↔ z←ω[(¯1+⍴ω)-⍳⍴ω]
```

It is not necessary to primitivize all expressions, if the transformation is obvious. In the above example, scalar-oriented languages cannot generate the vector resulting from $\iota\rho\omega$, but it is obvious how to write a loop to achieve the same result. Even though the example does have an obvious expansion, it is safer to take the process a step further, to precisely reflect the implementation language:

```
 yac←¯1 ⍝ yet another counter
I0: →((⍴ω)≤yac←yac+1)⍴Iz
 z[yac]←ω[¯1+(⍴ω)-yac]
 →I0
Iz:
```

The reason for this is apparent: An auxiliary variable and control structures have appeared. Since the author somehow always manages to code conditional branches and end conditions backwards, and gets counters initialized wrong, this technique is one of self-defense, to ensure that the errors appear at the modelling stage, not in the

implementation. In addition, bugs such as index errors, which are detected in APL, but not in many other languages, are caught here.

The detailed model should be run against the intermediate model (and the concise model if appropriate) to verify that they match. A discrepancy indicates an error in one of the models.

TRANSCRIPTION

Once the detailed model operates correctly, production of implementation language code should be little more than a transcription process. For example, the expression *i*←*i*+1, for *i* a scalar, might be transcribed into IBM S/370 assembler code as:

```
1          la     r0,1
i+1        a      r0,i
i←i+1      st     r0,i
```

Vetting of the implementation is performed by comparing the result of running it against the models, with identical arguments supplied to both systems.

If a suitably restricted dialect of APL is used for the detailed model, a compiler may be used for this phase. Since early 1983, the node computers used in IPSANET, I.P. Sharp's proprietary packet switching network, have been programmed in a subset of APL [Cr83], using a compiler (itself written in APL) developed by Steven Crouch, of the I.P. Sharp Network Development Department.

The time required to isolate faults can often be reduced by comparing intermediate results of the implementation against those of the model. The ability to step through an APL function in such cases, examining intermediate results, is exceptionally useful.

OPTIMIZATION AND COMPLEXITY ANALYSIS

APL models lend themselves to all sorts of optimization efforts. In the reversal example given previously, some simple optimization improves the inner loop:

```
 yac←⍴⍵
 k←yac-1
l0:→(0>yac←yac-1)⍴lz
 z[yac]←⍵[k-yac]
 →l0
lz:
```

The changes here include removal of some inner-loop arithmetic, and limit testing against zero, which is faster on certain machines than comparing two numbers. Similar techniques may be used to replace multiplication by addition, take advantage of hardware parallelism, move the limit test to the end to reduce inner-loop branches, etc.

More complex optimizations, visible at the concise model level, are often masked by code volume at lower levels. Consider a model of a loom for weaving cloth, which is a function of 3 arguments:

thread - A one-row integer matrix, specifying which harness a given thread is connected to.

treadle - An integer vector, specifying the treadling pattern used.

tieup - An integer matrix, specifying which harnesses are connected to which treadle.

The original model, approximately as presented to the author by a weaver [Po82], follows. To simplify the presentation, *tieup* is a global variable:

```
 z←⍺ weave0 ⍵;n
 z←((⍴⍺),⍴⍵)⍴⍵
 n←0
l1:z[n;]←⍵∈tieup[⍺[n];]
 n←n+1
 →(n<⍴⍺)/l1
```

A bit of analysis showed that this could be expressed as an inner product and some array indexing:

```
weave1: ' *'[tieup[⍺;]∨.=⍵]

tieup←4 2⍴0 3 2 3 1 2 0 1

(10⍴(⍳4),⌽⍳4) weave1 20⍴⍳4
*  **  **  **  **  *
  **  **  **  **  **
 **  **  **  **  **
**  **  **  **  **
**  **  **  **  **
 **  **  **  **  **
  **  **  **  **  **
*  **  **  **  **  *
*  **  **  **  **  *
  **  **  **  **  **
```

Computation of a 100 by 100 weave, using *weave*1, takes about 70 seconds on SHARP APL/PC. A similar function, written in BASIC, took several minutes, and required two and one half pages of code [He82]. Complexity analysis at this point is desirable and enlightening. *tieup* is typically small, of shape 8 8 or smaller, whereas *treadle* and *thread* are usually several hundred elements each. The function consists of two index operations, which are approximately linear with respect to result size, and an inner product which is nonlinear and of order *n*×3, where *n* is the approximate shape of *treadle* and *thread*. Since the arguments to the inner product are large in shape, a reduction in their size will likely reduce processor time by much more than optimization efforts directed at the index expressions.

Since (⍴*treadle*)>1↑⍴*tieup*, the left argument to the inner product (*tieup*[⍺;]) must contain duplicate rows. This implies redundant computation in the expensive inner product. If we instead compute all possible treadling results once, and then index those results with the actual treadling

pattern used, the time spent in the inner product is significantly reduced. This version takes 14 seconds on SHARP APL/PC:

```
weave2: ' *'[(tieupv.=⍵)[⍺;]]
```

The next candidate for optimization is the *character*[*boolean*] indexing operation. *weave*2 indexes the character vector ' *' with a Boolean matrix which contains many duplicate rows. If the indexing is factored out and performed once, the processor time required drops to 7 seconds, a factor of 10 improvement over *weave*1, and much faster than the BASIC version:

```
weave3: (' *'[tieupv.=⍵])[⍺;]
```

These sorts of optimizations may be obvious at the concise model level, but require significant amounts of careful analysis, measurement, and study at the detailed model or implementation level, simply because the great amount of detailed code masks the process being performed. Furthermore, attempts to optimize at the detailed model or implementation level are inevitably error-prone, again because of the code volumes involved at those levels.

HYBRID MODELS

Frequently, a prototype written strictly in APL is inadequate to model the solution to a problem. For example, the APL model of the semantic analyzer for a PL/I compiler might accept as input an intermediate file created by an existing parser, written in a language such as PASCAL. A terminal session manager which is intended to drive a real-world device may require extensive experimentation to make it work at all.

In such cases, a **hybrid model**, which might use shared variables to communicate with a simple auxiliary processor, may be of value. The structure of such a model is easily understood by looking at an actual example.

SHARP APL session managers communicate with APL via a shared-variable interface (AP1) to the interpreter, and communicate with the terminals they are managing via interfaces such as VTAM, provided by the operating system.

When Eric Iverson wrote SAPV [Bu81], an early IBM 3270 terminal session manager, he viewed the session manager in this way:

3270 - VTAM - TD - SH - AP1 - APL

The device (a 3270 terminal) communicates with VTAM, a terminal access method provided by the operating system. The session manager is divided into two parts: The first part is a terminal driver (TD) which is concerned with the details of actually driving a particular device, such as datastream generation and decomposition, session establishment with the operating system, and so on. The terminal driver communicates with VTAM on one side at a very primitive level and with the session handler (SH) on the other side, at a relatively high level. The second part is a session handler, which performs logical functions that are independent of device peculiarities such as scrolling, windowing, generic support of function keys, and so on. Most of the visible functionality of a session manager lies in the session handler.

In a classical programming environment, a programmer would write the entire session manager, then sit down and debug it via such time-honored techniques as instruction step and core dumps to examine datastreams and operating system return codes. This is a slow, hard way to determine the answers to questions like: Why did the screen scroll too far? Why does it ignore PF keys? Why doesn't reverse video work correctly?

The terminal driver frequently requires a lot of iterative, tedious work to understand what's really going on with the device. Core dumps are not the most user-friendly debugging mechanism ever invented, nor the fastest. If coding, recompilation, and relinking are required between successive tests, the speed of development will suffer.

A hybrid model to assist in the development of such an application might look like this:

3270 - VTAM - AUXP - MODEL - AP1 - APL

AUXP is a very simple auxiliary processor. It drives the device in the simplest possible way. It accepts a datastream from the APL model via a shared variable, and passes that directly to VTAM. It waits for the VTAM response, and passes that response back to the APL model. AUXP is effectively transparent, and acts only as the interface between the model (written in APL), and VTAM.

The model is a collection of functions to perform both session handling and terminal driver duties, communicating with AUXP on one side, and with the APL system on the other side (via AP1).

During development, it is convenient to have two terminals side by side. One terminal runs the APL model, and the other is the device being managed. Aside from response time, a user of the device being managed is unable to tell if the session manager is being run in APL or in some other language. All the facilities of the terminal are available.

If an abnormality occurs during use, such as incorrect scrolling, all the power of APL is available to isolate the problem, including:

- ability to trace and stop function execution
- display of variables in the model, such as datastreams
- ability to alter the model functions or data at any point
- ability of the model to halt itself when an error occurs

The device may be driven interactively by sending it a datastream and examining the result. The ability to generate and decode datastreams with ease using APL functions greatly simplifies the development process.

Once the model runs in a satisfactory manner, the methods described previously may be used to generate the implementation code, which should work as modelled.

COMPATIBLE MODELS

The shared-variable part of the auxiliary processor may either be replaced by the implementation code, or it may be retained to provide a compatible model including such facilities as:

- A switch to allow choosing between implementation code and model code, perhaps on a demand basis, for one particular device only. This allows "splicing" into a session, to chase a particular problem, and also supports continued development work on a running session manager, without affecting other users of that session manager.
- Optional monitoring of datastreams by a shared-variable partner, while the session continues to be managed by the implementation code. This comes in handy when examining hard-to-reproduce failures which arise in the course of running real applications.

These facilities, combined with a model which is kept current with the actual implementation, are also useful in supporting new device features, such as graphics, as they become available.

CODE RELIABILITY AND STRUCTURE

Code produced via the modelling process has proven to be more bug-free, and more "structured", than that produced via other mechanisms. One reason for this is that APL encourages a functional, rather than subroutine, view of the world. This in turn encourages clean, simple interfaces, and functions with well-specified arguments and results, and no side effects. This has a lingering effect, since such code is usually quite amenable to modification, enhancement, or complete replacement in the future, by someone other than the original author. The tendency in APL to write straight-line code also makes vetting of code easier. The minimization of complicated code paths simplifies the application of techniques such as code coverage to ensure that all instructions paths are executed, and enhances the value of such techniques.

In the implementation of high performance versions of the SHARP APL functions **rotate** and **reversal** in 1979, hierarchical modelling paid off in two ways: The new functions ran more than ten times faster than their predecessors, and only one bug has ever been reported. The bug was in an area which had not been modelled, because the model was not a complete model of APL, and the system integration was performed incorrectly.

LIMITATIONS OF PROTOTYPING

In practice, the picture isn't always quite as rosy as it's been painted here. There are a number of things which are impractical or impossible to model, and problems which are not detectable by modelling.

Race conditions in a TCMP (Tightly-Coupled Multi-Processor) environment are not caught by a naive prototype. This is an area where further research into prototyping may be of significant value.

Integration of partial prototypes into an existing system may not be trouble-free, as the author observed in 1979 with **rotate**. System integration remains a serious problem with partial models.

Concise models, more so than detailed models, may suffer from problems such as workspace full, when presented with actual data. Programmers, using hybrid and hierarchical prototyping to model the semantic analyzer for a PL/I compiler, encountered a hard workspace full in a concise model when it was used to compile a 10,000-line PL/I program.

The cost of converting a partial prototype to a full-feature prototype may outweigh any savings gained by doing so; particularly if the final implementation language is APL.

Unless special care is taken, machine or language-dependent constructs may cause problems. An APL model which sometimes executes $0 \div 0$ may appear to work correctly, but unless a very critical eye is cast upon the transcription process, the implementation may fail wher presented with the same arguments. These problems may be addressed by creation of a suite of functions to perform the desired operations. A divide function which properly handles $0 \div 0$ might be written as:

$$div1: \alpha \boxdot \omega$$

or as

$$div2: \alpha \div \omega \dashv \div \omega$$

In 1984, when Eugene McDonnell rewrote the elementary functions in SHARP APL for improved performance and accuracy, he began by writing APL functions which mimic, to the bit level, each of the S/370 floating-point instructions which he intended to use. Detailed models were built using these functions, and the resulting code was installed with minimal problems.

Finally, the speed of an APL model may limit its applicability in certain problems where raw speed is paramount. Current research into APL compilers may help to solve this particular problem, at some cost in ease of interaction.

PERFORMANCE ESTIMATION

APL models can be useful for estimating the performance of the resulting system, in several ways. As the section on optimization stated, it is usually easier to perform complexity analysis at the concise model level, than on the actual implementation. Such analysis is a good predictor of performance of an algorithm when presented with various arguments. For example, a naive implementation of the APL set membership function $\alpha\in\omega$, on characters or Booleans, might use an algorithm which picks an element from α and then performs a linear search for it in $,\omega$. This algorithm has complexity:

$$(\times/\rho\alpha)\times\times/\rho\omega$$

An algorithm [Be73] which first builds a table using elements of ω, and then indexes that table using elements of α, has complexity:

$$(\times/\rho\alpha)+\times/\rho\omega$$

If the arrays contain 100 elements each, the speed ratio of the two algorithms is about 50 to 1.

If the path length of the implementation code can be estimated from one of the models, and if monitors are written into the model to count the number of calls, then the processor time required to execute the implementation will be proportional to the path length times the number of calls.

If a model of the present implementation exists, then comparison of the execution time of the new model versus the old one may help to indicate relative performance. Some caution must be exercised here, to avoid problems with model performance anomalies due to differences in the APL primitives, rather than differences in the models. In the case of the models of **indexof** presented in [Be73], not only was the new model faster than the old one, but the new model frequently outperformed the APL primitive it was intended to replace. Such clearcut proof of superior performance is hard to beat.

The use of APL for test data generation and performance measurement often proves to be of value in unexpected ways. Figure 1 shows relative performance plots of the old and new **rotate** function, measured while the function was being rewritten for performance reasons. The rather odd spikes appearing in an otherwise linear graph puzzled the author, who searched code and model unsuccessfully for an answer. The realization that the spikes were sporadic, and apparently non-reproducible didn't help. When it became clear that they were in some way related to user load, and that the old code exhibited the same anomaly, the search widened.

Further study revealed that the spikes (consuming enough processor time to perform a 4,000-element integer rotate!) occurred when another APL user entered a line of input, interrupting the execution of rotate. The anomaly was traced to the "set storage key" instruction of the newly-installed mainframe processor, which, due to the non-store-through nature of the cache on the machine, took an excruciatingly long time to execute. The computer architect chose to not correct the problem, so a software fix was developed, which avoided use of the offending instruction. This resulted in the pleasant curves shown in Figure 2.

I'M SURE THIS WORKED LAST WEEK

One very important part of any modelling work is to keep track of the various changes which are made during the course of modelling. Sometimes, one makes changes to many parts of the model in an attempt to correct a problem which is later found to lie in a totally different area. If good records are kept of all changes to the model, it is relatively easy to back up to a previous model, compare models, and so on. The use of APL as a base for modelling allows a lot of the clerical work associated with such recordkeeping to be automated. For example, the consistent use of an APL application maintenance tool, such as LOGOS [Go86], which keeps an audit trail of all changes made to the source, makes it possible to revive old versions of the model with little effort, and to include commentary on why the change was made.

A CLOSING WORD

As a programmer who struggled for many years with core dumps, dirty bit switch contact, and burned-out CPU indicator lamps, the use of prototyping has been a great relief. It has reduced the count of sleepless nights, and has made system programming once more APL: A Pleasant Life.

ACKNOWLEDGMENTS

I owe a considerable debt of gratitude to my colleagues: to Roger Moore, who got me interested in prototyping in 1971; to Hiroshi Isobe, who requested that I give a talk on prototyping, and thereby forced me to formalize my thinking about it; and to Matsuki Yoshino, who helped me to see the value and limitations of applying these techniques to very large projects. Comments by Arlene Azzareilo, Leslie Goldsmith, and an anonymous APL86 referee resulted in substantial revision of the manuscript.

BIBLIOGRAPHY

[Be73] Bernecky, R., "Speeding up Dyadic Iota and Dyadic Epsilon", APL CONGRESS 73, North-Holland Publishing Company, Amsterdam, 1973 (Post-congress edition).

[Bu81] Burger, J.D. "SATN-37: IBM 3270 User Guide", I.P. Sharp Associates Limited, 1981.

[Cr83] Crouch, S., Private communication, 1983.

[Di68] Dijkstra, E.W., "The Structure of the 'THE' Multiprogramming System", Communications of the ACM, Vol. 11, No. 15, May 1968.

[Go86] Goldsmith, L.H., et al, "LOGOS User's Guide", I.P. Sharp Associates Limited, 1986.

[He82] Heiser, P., "A Weaving Simulator", Byte Magazine, September 1982.

[Po82] Powell, Denise A., Private communication, 1982.

[Wy13] Wyszkowski's Second Law: Anything can be made to work if you fiddle with it long enough.