FOREWORD

The field of digital simulation language, although barely ten years old, has shown a remarkable growth and vigor. The very number and diversity of languages suggests that the field suffers from a lack of perspective and direction.

While claiming no expertise in the writing of sophisticated compilers, the authors believe a relative unconcern with implementation details permits a wider objectivity in matters of format and structure. Competence to speak on these aspects is claimed on the basis of extensive analog, hybrid and simulation language experience.

In Locke's words, "everyone must not hope to be ... the incomparable Mr. Newton ... , it is ambition enough to be employed as an under-laborer in clearing the ground a little, and removing some of the rubbish that lies in the way to knowledge.""1

INTRODUCTION

The appellation "digital simulation language" unfortunately has been appropriated by two quite distinct fields: simulation of discrete, serial processes, as typified by the GPSS and SIMSCRIPT languages; and simulation of parallel, more or less continuous systems, as typified by the MIDAS or DYSAC languages. The scope of this paper is limited to the latter.

This field of what might be called parallel languages has enjoyed a vigorous growth since its inception ten years ago. New languages are appearing at frequent intervals, but it appears the effort is scattered, and perhaps needs to be channelized. The authors have applied themselves to providing a measure of needed direction and perspective.

BRIEF SURVEY OF THE FIELD

History

Since Selfridge's article appeared in 1955,2 the field of analog like simulation languages for digital computers has grown at a rapid rate. Brennan and Linebarger3,4 have provided an excellent review and analysis of the history of the field; their surveys are summarized and somewhat augmented below.

Lesh,5 apparently inspired by Selfridge's work, produced DEPI (Differential Equation Pseudo Code Interpreter) in 1958. Hurley6 modified this language for the IBM 704 (DEPI 4), and then in conjunction with Skiles7 wrote DYSAC (Digitally Simulated Analog Computer). This line of development has continued at the Universities of Wisconsin and Colorado under Professors Skiles and Ride-
out, resulting in the BLOC languages (MADBLOC, HYBLOC, FORBLOC and COBLOC).8,9

Stein and Rose, in addition to generating ASTRAL,10 (Analog Schematic Translator to Algebraic Language) in 1958, have provided the theoretical and practical background needed to write a sorting routine, i.e., an algorithm to deduce the proper order of problem statement processing.11 This feature, although overlooked by many authors, is one of the keys to a useful language. This point is detailed below.

In 1963, Gaskill et al.12 wrote an excellent simulation language, DAS (Digital Analog Simulator). The program unfortunately suffered from a rudimentary integration algorithm and the lack of sorting.* MIDAS13 (Modified Integration DAS), written by Sansom et al at Wright-Patterson Air Force Base, supplied the necessary improvements and met with unprecedented success. (Approximately 100 copies of the program have been distributed.) The success of MIDAS is explainable by two facts; the integration routine and sorting feature make it extremely easy to use and MIDAS was written for the widely used IBM 7090-7094. The authors of MIDAS have now offered another entry, MIMIC,14 which is implemented by a compiler program and provides symbolic labelling, logical control capability, and freedom from the block-oriented programming. MIMIC is not without faults, particularly in the areas of data entry, but seems destined for the same general acceptance as MIDAS—and deservedly so.

One of the most significant developments has been computer manufacturer interest in simulation languages. Scientific Data Systems has led the field in this respect, having proposed DES-1 (Differential Equation Solver) in 1962.15 DES-1 has been modified extensively in the succeeding years, and now offers one of the best formats and one of the most variegated operator lists;16 SDS has always promoted the language as part of a total computer system which provides analog type I/O and programming for a digital computer.

In late 1964, IBM entered the field in a small way with PACTOLUS, by R. D. Brennan.17 Apparently, PACTOLUS was intended to have only modest computational capabilities and to contribute primarily as an experiment in man-machine communication. In this respect, the objectives of PAC-TOLUS are similar to those of DES-1, although the latter also attempts to be as powerful a computational tool as possible.

PACTOLUS was written for the IBM 1620, and brought simulation to a previously untapped audience of small installations. The popularity of PACTOLUS is rivaled only by that of MIDAS, if indeed it has a rival.

More recently, R. N. Linebarger of IBM has announced DSL/90 (Digital Simulation Language for the 7094 class computer).18 This language is a significant advance over PACTOLUS as a computational tool and offers many format improvements.

The above are only a few of the languages; many others have appeared. Table 1 shows a list of simulation languages, along with some important characteristics of each. This table is the result of a rather diligent search, but certainly is not comprehensive. Much of the material has been taken from a survey given by Clymer.19

Trends

The present trend in the field is towards extension of the power and utility of the programs. More efficient execution has been recognized as a goal, and the compiler approach to implementation has gained increased acceptance. The provision of more complex operators and more advanced algebraic capability represents an effort to increase the utility of the programs for less analog-oriented applications. These trends are in the right direction, but efforts have been scattered, and perhaps need to be channelized.

Another major step has been recognition of the importance of the man-machine interaction. As was mentioned, this has been the primary message carried by the PACTOLUS program, and has long been the concern of the DES-1 designers. The SCADS20 program has been used on-line at Carnegie Institute of Technology and the Aerospace Corporation is using EASL21 through a terminal to an IBM 7094. The authors agree wholeheartedly with this stress on man-machine interaction, and are aware of the real import this communication has for the future of digital simulation.

Areas of Use

Simulation languages have been written for two more or less diverse reasons: to provide analog
<table>
<thead>
<tr>
<th>Name</th>
<th>Source of Name</th>
<th>Date</th>
<th>Author(s)</th>
<th>Affiliation</th>
<th>Computer</th>
<th>Integration Routines</th>
<th>Ancestor Sorting</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1955</td>
<td>R. G. Selfridge</td>
<td>USNOTS</td>
<td>IBM 701</td>
<td>Simpson's Rule</td>
<td>No</td>
<td>The Adam of this genealogy</td>
</tr>
<tr>
<td>DEPI</td>
<td>Diff. Eq. Pseudo</td>
<td>1957</td>
<td>F. Lesh</td>
<td>Jet Propulsion Lab</td>
<td>Burroughs 204</td>
<td>4th Order Runge-Kutta</td>
<td>Selfridge No</td>
<td>Expanded and improved Selfridge's work</td>
</tr>
<tr>
<td>DIDAS</td>
<td>Digital Differential</td>
<td>1957</td>
<td>G. R. Slayton</td>
<td>Lockheed-Georgia</td>
<td>IBM 704</td>
<td>Euler</td>
<td>——</td>
<td>Simulates a DDA</td>
</tr>
<tr>
<td>ASTRAL</td>
<td>Analog Schematic</td>
<td>1958</td>
<td>Stein, Rose and Parker</td>
<td>Convair Astronautics</td>
<td>IBM 704</td>
<td>Runge-Kutta</td>
<td>Yes</td>
<td>Isaiah, the voice that crieth in the wilderness. A precursor of the modern languages. Sorting and compiler implementation were original with ASTRAL, and remained advanced features until very recently.</td>
</tr>
<tr>
<td>DEPI-4</td>
<td>DEPI for the</td>
<td>1959</td>
<td>J. R. Hurley</td>
<td>Allis-Chalmers</td>
<td>IBM 704</td>
<td>4th order Runge-Kutta</td>
<td>DEPI No</td>
<td>First language to use float point hardware, and thus eliminate scaling problems.</td>
</tr>
<tr>
<td>DYANA</td>
<td>DYnamics ANAlyzer</td>
<td>1959</td>
<td>T. J. Theodoroff</td>
<td>General Motors</td>
<td>IBM 704</td>
<td>Euler</td>
<td>——</td>
<td>Mechanical system dynamic analyzer</td>
</tr>
<tr>
<td>BLODI</td>
<td>BLOck DIagrammed</td>
<td>1961</td>
<td>Kelly, Lochbaum, and Vyssotsky</td>
<td>Bell Labs</td>
<td>IBM 704  and 7090</td>
<td>None</td>
<td>—— No</td>
<td>Block simulator for signal processing devices</td>
</tr>
<tr>
<td>DYSAC</td>
<td>Digitally Simulated</td>
<td>1961</td>
<td>J. J. Skiles and J. R.</td>
<td>Univ. of Wisconsin</td>
<td>CDC 1604</td>
<td>4th order Runge-Kutta</td>
<td>DEPI-4 No</td>
<td>The prophet with honor only in his own country. Significant improvement of DEPI-4, particularly in format. The program was specific for a relatively little used computer, which probably caused its undeserved lack of wide acceptance and use outside the University of Wisconsin.</td>
</tr>
<tr>
<td>DYNASAR</td>
<td>DYNamic Systems</td>
<td>1962</td>
<td>Lucke, Robertson and Jones</td>
<td>General Electric-Evendale</td>
<td>IBM 704  and 7090</td>
<td>Adams-Moulton 4 point predictor-corrector Variable integr, step-size</td>
<td>Yes</td>
<td>Useful innovation was variable step size integration algorithm.</td>
</tr>
<tr>
<td>PARTNER</td>
<td>Proof of Analog Results Through Numerically Equivalent Routine</td>
<td>1962</td>
<td>R. F. Stover</td>
<td>Honeywell Aeron. Div. and H-800/1800</td>
<td>IBM 650</td>
<td>IBM 7900</td>
<td>Trapezoidal or Euler</td>
<td>No but retains parallelism</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------------------</td>
<td>-----</td>
<td>-------------</td>
<td>----------------------------------</td>
<td>--------</td>
<td>---------</td>
<td>---------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>DAS</td>
<td>Digital Analog Simulator</td>
<td>1963</td>
<td>R. A. Gaskill</td>
<td>Martin-Orlando</td>
<td>IBM 7090</td>
<td>IBM 7090</td>
<td>DYSAC No</td>
<td>Major contributions to the format of block-oriented languages. Widely used in the Martin Company.</td>
</tr>
<tr>
<td>JANIS</td>
<td>?</td>
<td>1963</td>
<td>R. G. Byrne</td>
<td>Bell Labs</td>
<td>IBM 7090</td>
<td>Euler</td>
<td>No</td>
<td>FORTRAN flavor</td>
</tr>
<tr>
<td>DES-1</td>
<td>Differential Equation Solver</td>
<td>1963</td>
<td>M. L. Pavlevsky and L. Levine</td>
<td>Scientific Data Systems</td>
<td>SDS 9300</td>
<td>Choice of five</td>
<td>No</td>
<td>Excellent language. Part of a computer system that includes a special, analog type console.</td>
</tr>
<tr>
<td>DIAN</td>
<td>Digital Analog Simulator</td>
<td>1963</td>
<td>Farris and Buckhart</td>
<td>Iowa State Univ.</td>
<td>IBM 7074</td>
<td>Euler</td>
<td>No</td>
<td>Chemical Engineering Simulations</td>
</tr>
<tr>
<td>COBLOC</td>
<td>CODsp Language BLock Oriented Compiler</td>
<td>1964</td>
<td>Janoski and Skiles</td>
<td>Univ. of Wisconsin</td>
<td>CDC 1604</td>
<td>Choice of three</td>
<td>DYSAC Yes (Optionally No)</td>
<td>Logical building blocks (gates, flip-flops), etc. provided.</td>
</tr>
<tr>
<td>FORBLOC</td>
<td>FORTRAN compiled BLock Oriented Simulation Language</td>
<td>1964</td>
<td>W. O. Vebber</td>
<td>Univ. of Wisconsin</td>
<td>Any machine</td>
<td>Trapezoidal DYSAC No</td>
<td>No</td>
<td>Easily modified since FORTRAN used. This approach could lead to machine independence.</td>
</tr>
<tr>
<td>HYBLOC</td>
<td>HYbrid computer BLock Oriented Compiler</td>
<td>1964</td>
<td>J. R. Hurley</td>
<td>Allis-Chalmers and Univ. of Wisconsin</td>
<td>IBM 709, 7090, 7094</td>
<td>4th order</td>
<td>DYSAC No</td>
<td>Simulates hybrid computer.</td>
</tr>
<tr>
<td>MADBLOC</td>
<td>MAD Language BLock Oriented Compiler</td>
<td>1964</td>
<td>L. Tavernini</td>
<td>Univ. of Colorado</td>
<td>IBM 7090</td>
<td>Trapezoidal DYSAC No</td>
<td>No</td>
<td>MAD (Michigan Algorithmic Decoder) statements.</td>
</tr>
<tr>
<td>MIDAS</td>
<td>Modified Integration DAS</td>
<td>1964</td>
<td>Harnett, Sansom and Warshawsky</td>
<td>Wright-Patterson AFB</td>
<td>IBM 7090-7094</td>
<td>5th order</td>
<td>DAS Yes</td>
<td>The Moses of the story, that led digital simulation to the verge of the Promised Land. Very widely distributed, modified and discussed.</td>
</tr>
<tr>
<td>SIMTRAN</td>
<td>---</td>
<td>1964</td>
<td>W. J. Henry</td>
<td>Weapons Research Establishment — Australia</td>
<td>IBM 7090</td>
<td>---</td>
<td>DAS Yes</td>
<td>Though not used on-line, the program's structured for such use in the future.</td>
</tr>
<tr>
<td>Model</td>
<td>Year</td>
<td>Author(s)</td>
<td>Language</td>
<td>Computer</td>
<td>Order</td>
<td>Architecture</td>
<td>Main Features</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-----------</td>
<td>----------</td>
<td>----------</td>
<td>-------</td>
<td>---------------</td>
<td>---------------</td>
<td></td>
</tr>
<tr>
<td>PACTOLUS</td>
<td>1964</td>
<td>R. D. Brennan</td>
<td>IBM Research Lab</td>
<td>IBM 1620</td>
<td>2nd order</td>
<td>MIDAS Yes</td>
<td>Mainly an experiment in man-machine communication. Widely used as a simulation tool.</td>
<td></td>
</tr>
<tr>
<td>ENLARGED MIDAS</td>
<td>1964</td>
<td>G. E. Blechman</td>
<td>NAA-S&amp;ID</td>
<td>IBM 7090</td>
<td>Same as MIDAS</td>
<td>MIDAS Yes</td>
<td>Enlarged component set of MIDAS and added plotting routines.</td>
<td></td>
</tr>
<tr>
<td>PLIANT</td>
<td>1964</td>
<td>R. L. Linebarger</td>
<td>IBM Develop. Lab.</td>
<td>IBM 7090</td>
<td>Trapezoidal JANIS</td>
<td>No</td>
<td>Build own FORTRAN blocks.</td>
<td></td>
</tr>
<tr>
<td>MIMIC</td>
<td>1965</td>
<td>F. J. Sansom</td>
<td>Wright-Patterson AFB</td>
<td>IBM 7090-7094</td>
<td>4th order</td>
<td>MIDAS Yes</td>
<td>Improvements over MIDAS include: compiler implementation, logical elements, improved algebraic capability, and logical control.</td>
<td></td>
</tr>
<tr>
<td>DSL/90</td>
<td>1965</td>
<td>Syn and Wyman</td>
<td>IBM Develop. Lab.</td>
<td>IBM 7090</td>
<td>Choice of eight</td>
<td>PLIANT Yes (Optionally No)</td>
<td>Powerful, flexible simulation tool. Advanced format ideas include free format capability.</td>
<td></td>
</tr>
<tr>
<td>SCADS</td>
<td>1964</td>
<td>J. C. Strauss and W. L. Gilbert</td>
<td>Carnegie Tech</td>
<td>CDC G-20</td>
<td>PartNER No</td>
<td>SCADS. Uses the same scheme for parallel operation as PARTNER, i.e., an extrapolation method. SCADS was used on-line at Carnegie Tech.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EASL</td>
<td>1965</td>
<td>L. Sashkin and S. Schlesinger</td>
<td>Aerospace Corp.</td>
<td>IBM 7094</td>
<td>4th order</td>
<td>MIDAS No</td>
<td>Used on-line through a terminal. FORTRAN statements are permitted in line.</td>
<td></td>
</tr>
<tr>
<td>SADSAC</td>
<td>1965</td>
<td>J. E. Funk</td>
<td>U.S. Air Force Academy</td>
<td>Burroughs B-5000 and B-5500</td>
<td>5th order</td>
<td>MIDAS Yes</td>
<td>Essentially MIDAS, but written in ALGOL. Significant feature is handling of discontinuities by interrupting integration routines when switching occurs.</td>
<td></td>
</tr>
<tr>
<td>SLASH</td>
<td>1965</td>
<td>J. E. Funk</td>
<td>U.S. Air Force Academy</td>
<td>Burroughs B-5000 and variable step SACK</td>
<td>5th order</td>
<td>SACK Yes</td>
<td>Gives an ALGOL program control of SACK for parametric studies, plotting optimization, etc.</td>
<td></td>
</tr>
</tbody>
</table>
check cases, and to solve differential equations (in other words, replace an analog computer). MIDAS, ASTRAL and PARTNER were written primarily for the first reason; DAS, the DEPI family, DES-1, and PACTOLUS, apparently for the second. Another, perhaps more important, use has not been stressed by any of the authors, i.e., providing the best digital computer language for a hybrid problem. Since an analog computer is a parallel device, and no amount of mental calisthenics can make it appear serial, it is clear that a parallel digital language is the only solution to the parallel-serial dichotomy in hybrid programming.

FUNDAMENTAL NATURE OF SIMULATION LANGUAGES

Views Proposed

As noted above, simulation languages have proliferated at an amazing rate in the last few years. Each new language comes equipped with its own format and structural idiosyncrasies, which generally reflect the creator's reading of the essence of simulation languages. These analyses might be classed as: analog computer simulator, block diagrammed system simulator, and differential equation solver. When the concepts are considered in detail, it is evident that all these views miss the point to some extent.

Analog Computer Simulator: Some simulation languages, as noted previously, were written specifically to simulate an analog computer. The purpose was to provide an independent check of the analog setup. Most of the authors state, however, that the resultant programs were used profitably for problem solving—in other words, the analog was bypassed.

Simulating the analog computer generally results in an operator (or statement) repertoire which reflects the fundamental physical limitations of analog computer elements. Examples of this phenomenon abound; one might mention the limitation on the number of summer inputs, different elements for variable and constant multiplication, and the lack of memory.

In short, the logical culmination of this concept is a system neither fish nor fowl, with many disadvantages of both analog and digital programming.

Block Diagrammed System Simulator: The overwhelming majority of authors state that their language (or program) was designed to simulate systems that can be represented by block diagrams. (Of course, an analog computer is such a system, so the opinion outlined in the previous subsection can be seen to be a sub-set of this view). By and large this opinion is justified, since many problems of interest are easily expressible in block form. However, if the problem is given simply as a set of ordinary differential equations, reducing the equations to a block diagram is a tedious, error-prone process. Even if a block-diagrammed system is considered, more often than not some of the blocks contain differential equations; an example is the airframe equations in a control loop.

Differential Equation Solver: An opinion sometimes expressed is that simulation languages should be designed to solve ordinary differential equations. From what has been said, this view has some merit. Unfortunately, two problems arise. First, in control system simulation, transfer functions and nonlinearities are not conveniently expressible in equation form. There is also a certain loss of familiarity with the system when blocks are completely eliminated. Second, the concept overlooks other important problem areas, e.g., sampled data systems, where the problem is stated in difference equation form.

A More Correct Approach

It is seen, then, that all these views regarding the fundamental nature of simulation languages are too narrow and confining. Is there an "essence" (in the metaphysical sense) which is common to all, yet not so comprehensive as to be meaningless? Parallelism, the apparent parallel operation of a serial digital computer, may be an all-inclusive, rational statement of the essential nature.

All languages extant have taken their format and structural cues from analog programming and analog operators. Assuming the action was rational, and not an empty exercise in dialectical synthesis, this fact provides a clue to a valuable overall view of simulation languages. The analog computer is, of course, a parallel device.

As it happens, this is the way most of the world is structured. Representing physical phenomena with a serial digital computer is an artifice; useful, but nevertheless an artifice. The analog computer has achieved such success and generated such attachment largely because of the close analogy existing between the computer and the physical world.
Obviously, then, if physical systems must be represented with a serial digital computer, the machine should be made to appear parallel. This is in fact what has been done in simulation languages, and, of course, the success is manifest. Difficulties have arisen, though, because "pseudo-parallel" devices in the past been modeled too closely on the analog computer. If the notion of parallelism is correct, the best parallel device must be sought.

The importance of this concept cannot be overstated. It is not merely a convenient catch-all to include all previous efforts, but has real consequences for the future of simulation languages. A programmer if he is to “think parallel” must be freed from the chore of ordering problem statements. Such freedom is available if a sorting algorithm, as first proposed by Stein and Rose, is used. Alternatively, an extrapolation scheme, as used in PARTNER and SCADS, achieves the desired parallelism, but at a cost in storage and execution efficiency. The languages incorporating sorting or extrapolation are true parallel languages and provide the designer with a parallel device to represent his parallel physical system.

It is always treacherous to be dogmatic, but on this point it seems clear that a language without sorting (or its equivalent) is simply another, perhaps slightly superior, method of programming a digital computer, and is in no way a parallel system simulator.

**FORMAT**

**Present Format Inadequacies**

Perhaps the most important consideration in designing a simulation language is the utility of the input format. A good, flexible, natural appearing format would encourage wide usage, facilitate training, and reduce errors. All existing formats are much too arbitrary and generally reflect both the artificialities of digital computer modes of thought, and the physically determined inadequacies of analog elements.

Under digitally derived restrictions, one might mention the exaggerated importance given to column position in a statement, the need for commas and decimal points where unnecessary for clarity, and the requirement for a specific, arbitrary order of arguments within a statement.

Analog inadequacies have been mentioned; they appear as restrictions on the number of inputs to an element, poor logical and memory features, and rudimentary labelling capability. (This latter is a curious anachronism, since even the most primitive digital computer assemblers permit symbolic labelling.) Some formats, notably ASTRAL, were based directly on a specific analog computer and may be expected to have certain deficiencies. In others, e.g., MIDAS and PACTOLUS, no real attempt was made to simulate an analog computer yet the implied hardware limitations are nonetheless present.

As a consequence of the poor format, operational difficulties are found to stem from trivial clerical errors, such as dropping commas or decimal points, or having input statements in the wrong order. These difficulties are increased by the multiplicity of primitive operators, and the consequent need for large complex networks to represent algebraic statements. The artificialities also tend to make the language more difficult to learn, or having been learned, to retain all the esoteric details. (The retention of these details might seem a small matter to the "professional" programmer, but is a real concern to the occasional user.) Modern computers with character handling ability and high execution speed can free the programmer from this sort of detail, with very little penalty in increased compilation time.

**General Format Rules**

There is now a large fund of experience in the design and utilization of parallel languages, and some general, somewhat dogmatic, statements can now be made about format. On a very general level, these could be reduced to two rules: The format should be both “natural” and “non-arbitrary.” These rules require some amplification.

“Naturalness.” The input problem statement should be as close as possible to the normal, accepted method of problem statement. The question in the designer’s mind should be: if I were preparing a problem for my own future reference and explanation to others, how would I state it? The answer to such a query would naturally vary with the type of problem.

If a parallel control system is to be analyzed, the problem would most naturally be stated in a block diagram, wherein each block represented a more or less complex operator such as gain, limit, hysteresis, transfer function, etc.

If, however, a set of differential equations is to
be studied, it is a great deal of wasted effort to formulate the problem in block form. The process is conducive to error and really adds nothing to the understanding of the problem.

It is imperative that the input statement, the cards presented to the digital computer, should match as closely as possible the normal, natural statement of the problem. Of course, there are fundamental limitations, notably the fact that superscripts and subscripts are normally used while the card punch must work on one line. However, much can be done, and actually has been done to "naturalize" format, especially in the APACHE\textsuperscript{a} and UNITRAC\textsuperscript{b} programs.

To reiterate, the input statement should be flexible enough to match the natural form of diverse problems; block diagram and differential equation types have been mentioned. Many problems arise that are really combinations of these classes, e.g., an airframe in an autopilot loop. The format should naturally be capable of stating each problem area in its normal form.

Non-Arbitrariness: Arbitrary formats, more than anything, have limited the utility of parallel languages. One commonly finds early enthusiasm for the idea of parallel languages, and then disenchantment when the "format gap" between the idea and its implementation is fully appreciated. In many facilities, where digital computer turn-around is measured in days, the trivial errors caused by the complex, arbitrary format extend a problem's check-out time to unacceptable periods.

However, this is not a fundamental problem; surely parallel languages can be written to eliminate most of these difficulties. It seems that few authors have given much thought to minimizing arbitrariness in their concern for other aspects of the language. This is seen clearly in the early work of Selfridge and Lesh, where the root idea of parallel languages was the main subject of study. Unfortunately, examples are still apparent: Many outstanding contributors, in their understandable enthusiasm for man-machine interaction and efficient implementation, have been satisfied with adopting earlier format ideas and have neglected the ramifications of a good, clear, non-arbitrary input statement.

A few obvious requirements are discussed below:

(a) A "free format" statement capability, i.e., statements can appear anywhere on the card. This is available in the UNITRAC and DSL/90 programs, and should eliminate much of the frustration produced by coding or key punch slips.

(b) The ability to enter numerical data in any convenient form, e.g., 200.0 might be written 200, 200.0, 200., 2E2, 2.0E02, etc.

(c) The ability to use either literals in a statement (Y = 4X) or the option to symbolically label constants (Y = KX). In the latter case, the constant would be specified in the normal fashion. (K = 4 or K = 4., etc.)

(d) The ability to label quantities in a sensible, problem related fashion, and use this label to specify the variable without reference to an arbitrary block number. The latter labeling method could be retained for meaningless intermediate quantities. It should be noted that the need for problem related labeling was one of the first lessons learned by software designers and is now available with virtually all assemblers.

Complexity of Format

An open question at this time is the allowable degree of complexity in the input format statement. The trend in new languages appears to be away from the simple, analog type blocks to statements reminiscent of FORTRAN.

Table 2 shows an "algebraic capability" scale, with some of the languages distributed along it. At the lower end, one finds a very primitive capability, which can represent any algebraic statement, albeit in an extremely awkward form. Fortunately, no one has been inspired to implement this sort of language. (This is not to say such codes are useless; a primitive language is generally the intermediate representation in a compiler program.) Moving up the table, the next stage is basic mathematical operators modeled by and large on analog computer components. DAS and MIDAS are good examples of this class. Here, there is some advance from a "minimum vocabulary" and a great deal of flexibility is available, particularly for block oriented sys-

\textsuperscript{a}The APACHE\textsuperscript{24} program is not really of the genre under consideration here. The program takes differential equations and generates an analog wiring diagram and static check. However, the format considerations are nearly identical to equation solving languages, and the APACHE authors have produced an input format worthy of study.
Table 2. Algebraic Capability Scale.
(listed in order of decreasing capability)

<table>
<thead>
<tr>
<th>Description</th>
<th>Examples</th>
<th>Typical Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statements in any form understandable by the engineer</td>
<td>—</td>
<td>(\frac{dy}{dt} = y \sin wt + x^2)</td>
</tr>
<tr>
<td>Nested sum of products and any functions or operators</td>
<td>—</td>
<td>(y_0 = 5, y_0 = 0, w = 2\pi)</td>
</tr>
<tr>
<td>Nested sum of products and operators</td>
<td>MIMIC</td>
<td>(Z = (XY + K1 \sin WT) \times \text{ERF} )</td>
</tr>
<tr>
<td>Nested sum of products and certain functions (a la FORTRAN)</td>
<td>DSL/90, UNITRAC</td>
<td>(Y = \text{ADD}(X, \text{MPY}(B, Z, \sin(U))))</td>
</tr>
<tr>
<td>Nested sum of products</td>
<td>DES-1</td>
<td>(Y = X \times Y \times (\sin(A + B)) + K \times M)</td>
</tr>
<tr>
<td>Single level sum of products</td>
<td>ASTRAL, DEPI, DYSAC</td>
<td>(Y = K_1 \times X \times X + K_2 \times Z + K_3 \times K_4 - K_5)</td>
</tr>
<tr>
<td>Coefficients on inputs to block operators</td>
<td>DAS, MIDAS</td>
<td>(NO_4 = PO_1 \times NO_1 + PO_2 \times NO_2)</td>
</tr>
<tr>
<td>Basic mathematical operator (a la analog)</td>
<td>—</td>
<td>(MI: S_1, I_2)</td>
</tr>
<tr>
<td>Primitive operators, the minimum necessary, with the minimum inputs</td>
<td>—</td>
<td>(S_1: K_1, I_2, M_3, K_5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(S_1: K_1, I_2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(S_1: K_1, I_2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(I_2: K_1)</td>
</tr>
</tbody>
</table>

It is seen here that the evolutionary movement up the table is not in strict chronological order. ASTRAL and DEPI preceded DAS and MIDAS.

The next stage provides coefficient setting on all inputs to blocks of the type discussed above. As on an analog computer, only constants can be multiplied by the input variable.

A natural extension of coefficient setting is, of course, variable multiplication at element inputs, the next step up the table. (Division is also assumed permissible here.) This stage has not been implemented, probably because the transition to the next stage is so evident.

In this stage, exemplified by DES-1, nesting of sums of products (and quotients) is allowed, i.e., any level of parentheses is permissible. A flexible statement is provided, although no functional relationship (sin, exp) can be imbedded in the sum of products.

This lack is provided at the next stage, now available in the DSL/90 and UNITRAC languages. Here, an input statement very similar to FORTRAN is available; a limited number of functions may be used in the statement.

It might seem odd that a MIMIC type language, which allows only operators in the statement, should be set above UNITRAC or DSL/90 which permit functions. However, since function generators can be considered as operators, this format is extendable downward, and further allows operations like integration and limiting to be imbedded in the input statement.

Moving now to the top, the next stage provides a synthesis of the two below, allowing both operators and functions to appear in the nested sum of products. The allowable function list would be open ended and at the user's discretion. No published work provides this capability. At the very top, and really hardly in sight, is the capacity to accept any reasonable problem statement understandable to the engineer.

There certainly are objections to considering upward movement on the table as evolutionary, with the implied value judgment concomitant to that view. The authors would agree that a powerful algebraic capability, although very useful for some tasks, must at the present state of the art carry with it increased complexity and arbitrariness. The tool may prove too powerful for many users and lead to confusion and errors. Further, many users prefer the block approach and with excellent reasons.

However, all fears can be allayed, since regardless of the available complexity, lower level statements are possible by simply limiting the size and complexity of the more powerful statement. A close examination of Table 2 will show that any of the formats are easily derived by restricting the extent of those above.

Diagnostics

In general, program error diagnostics should be extensive and specific. Each card containing the error, and only that card, should be printed, along with a specific comment on the trouble. Alternatively, diagnostics could be printed immediately adjacent to the erroneous statement, as the source language is being printed. Closed algebraic loops.
should naturally be printed separately from format errors. It is expected that improved format will reduce this sort of error, but some are sure to appear and rapid checkout requires good diagnostics.

Diagnostics and corrections to the program must be permitted in the source language to free the programmer from the details of debugging at the machine code level. For hybrid work it is essential that such source level debugging and modification be permitted on-line through a console typewriter or similar unit. Extremely rapid compilation is of course necessary to make this economical.

On-line debugging and modification has been explored more fully in an earlier article by the authors. The particular framework in point was a huge multiprogrammed digital computer with analog type terminals (one might say a large scale multiprogrammed PACTOLUS, or a time shared DES-1). With such a system and a parallel language, the advantages of analog computers (parallelism, intimate man-machine communication) and digital computers (accuracy, repeatability) are both apparent. The language employed with this system must permit clear diagnostics and simple modification at the source level to retain the analog virtues of simple communication and rapid modification.

STRUCTURE

Integration with Software System

As is well known, all modern software systems contain many languages; one might mention FORTRAN, assembly language, ALGOL and COBOL. These languages are generally under the control of a monitor or executive program, which calls programs to compile (or assemble) various source programs into machine code. Usually subprograms can be written in any of the source languages, and a complete program linked at load time.

The parallel language should come under this organization, and be available along with FORTRAN and the rest, as the optimum source code for a particular class of problems. In this way, and using the subprogram feature, each problem area could be written in the best language for a particular task; say, parallel language for the differential equations, FORTRAN for the arithmetic, and machine language for chores such as masking, character handling, and Boolean operations.

Extension Capability

In order to remain useful in the face of continuously expanding user requirements, any language must be able to grow with needs. When a parallel language is examined with the intent of increasing capability without organizational disruption, it is seen that expansion should take the route of adding operators or functions. Expansion must, of course, fit neatly into the total software system, i.e., the other languages and the executive program, as outlined above. Since a common subroutine format is already available with the other languages, any subroutine, written in any source code, could be used by the parallel language programmer and be called simply by using the subroutine name as an operator.

Augmenting the operator set must be made very simple and obvious so the average user, unfamiliar with normal digital techniques, can exploit the extension feature without recourse to a systems or maintenance programmer.

User-Oriented Organization

The language should be designed to easily match the capabilities of diverse programmer levels. Basic subsets, such as primitive operators, algebraic statements, etc. should be made available to the less sophisticated programmers. These subsets should be capable of integration and mixed use by the more highly skilled user. Also, elements of a more complex nature (e.g., serial operators) should be available to the expert, but not a matter of concern for the novice. Thus, a structure is required that will permit the novice to learn a minimum subset and then advance, if he wishes, to the use of an extremely complex and powerful simulation language. (Or looking at it yet another way, there should be open and closed shop versions; the various open versions upwardly compatible with the closed shop version.)

In sum, there seems to be no need to restrict the language's use to a particular programmer level, if the initial design is done in a systematic manner.

IMPLEMENTATION

Regardless of format and structure, the language's effectiveness will depend entirely on the quality of
the implementation. This aspect has recently been a major interest area, and the concepts are becoming rather well developed.

In general, a program that produces machine code is a necessity for efficient execution. MIMIC and SCADS, compilers, directly achieve this, while DSL/90 and ASTRAL generate a FORTRAN deck which can then be compiled into an object program. This latter approach, (if a good FORTRAN compiler is used), can produce efficient code by exploiting the considerable efforts expended by FORTRAN designers. There are certain applications, particularly those with small machines, where an interpreter program makes more sense, but generally a compiler seems the best route. This is detailed more fully below. First, an examination of the trade-offs involved in writing a compiler program for a parallel language.

Compiler for Different Applications

As was mentioned, there are three major usage areas for parallel languages: analog check cases, differential equation solving, and the digital proton of a hybrid problem. The relative weights given to compiling and execution times vary with the particular application.

Analog Check Cases: This usage is generally on a single job basis, i.e., the program is compiled and run once and then discarded. Since the object program is never used again, only the sum total of compiling and execution time for one run need be minimized. In fact, this minimization is hardly a point to stress, since analog check cases would probably represent a small total of a digital facility’s work load.

All Digital Simulation: If the language is to be used for this application on a “load-and-go” basis, minimization of the total time is of prime importance. On the other hand, if production programs are the expected rule, execution time is the quantity to be minimized.

Hybrid: Here, the requirement for an efficient object program is a vital consideration, and real sacrifices can and must be made in compiling efficiency.

As a general rule, compiling time should never be minimized at the expense of input format, and only as a last resort should format be sacrificed for decreased execution time. This latter seems a remote possibility, but it is easy to see compiling time increased in the interests of simpler programming.

Different Computers

If this language is to achieve the general usage typical of FORTRAN, some thought must be given to implementation for diverse computers. It is pointless to design a system workable only for a CDC 6600-6800 or the top of the IBM 360 line. Similarly, it is a waste of effort to aim at implementation solely for a PDP-8, DDP 116 or SDS 92. The large machines obviously should have the full language, i.e., all subsets, and be provided compiled versions. For the smaller machines, two approaches are possible. The basic subsets could be compiler versions, thus providing efficient programs although only for small problems and at a modest language level. Alternatively, the complete system could be run in an interpreter mode, sacrificing time, but permitting the use of a very powerful tool on a small machine.

REQUIRED FEATURES

Along with implementation, the operational features of the language (the programmer’s bag of tricks) have been a major concern of language designers. This section does not aim to be all-inclusive; probably some of the “required” features have not been invented yet. The requirements can be subdivided into two classes: structural or logical features, and the types of elements or operators. Sorting is not discussed, since it is assumed that all modern parallel languages will be so equipped.

Structural Features

1. Logical Control: Logical control over program structure and execution is of paramount importance. DES-1, being sequential, easily incorporates this feature by the use of “IF” statements similar to FORTRAN. MIMIC, a true parallel language, still provides decision capability with the “logical control variable.”

In substance, statements or operators are serviced or bypassed as a function of problem variables. So long as the by-passing is done in the true digital sense (non-execution), and not the analog sense (execution, but no use made of the outputs), a substantial time savings is realized.
Logical control is quite important. Without it a parallel language yields no more than a hyper-accurate and hyper-repeatable analog computer; some of the best features of the digital computer, decision capability and memory, are unused.

2. **Multiple Rates:** This important provision, available in DES-1, minimizes execution time by servicing slowly changing variables only as required. Generally speaking, multiple rates increase the effective bandwidth of the simulation program. This has real import for hybrid work.

The multiple rate option is clearly a part of logical control capability. In this case, sections of the program are not executed at every pass, but unlike full logical control, the bypassing is not under control of a program variable.

3. **Macro Capability:** The macro capabilities of modern assemblers should be available to the parallel language programmer. Using this feature, prototypes of often used problem sections could be coded with unspecified parameters, and then subsequently used as integrated units. Macros would obviate repetitious coding of identical loops or problem areas, e.g., parallel sections of control systems that are identical in structure.

4. **Subprograms in Other Source Code:** In a normal digital computer operation, there is always a large library of routines available to the programmer. These programs should be easily incorporated within the parallel language. If expansion is implemented as suggested (see Extension Capability), not only would the entire library be available, but the programmer with digital training could use whatever language desired for particular problem areas. For example, logical or Boolean operations would be most easily handled in machine language.

5. **Repetitive Operation and Memory:** It should be possible to repeat runs, as on a repetitive analog computer, with new parameters calculated from previous runs. This implies two further requirements: function storage, and algebraic calculations between runs.

### Elements

In this section, the normally found operators, e.g. summers, multipliers, etc., are taken for granted. No attempt has been made to be comprehensive; however, those discussed are considered important and/or relatively rare in present languages.

1. **Integrator:** An accurate, efficient integration method is the *sine qua non* of digital simulation languages. Apparently no firm conclusions have been reached as to the best algorithm; the number of schemes tried is almost as large as the number of languages (See Table 1). As an example of the dynamics of this situation, note that Sansom, having used an excellent method (4 point variable step predictor-corrector) in MIDAS, changed to another (modified Runge-Kutta) in MIMIC.

DES-1, DSL/90 and COBLOC permit a number of integration options, ranging from simple Euler integration to complex Runge-Kutta and Adams-Bashford algorithms. This variety does allow the user to select a scheme which is adequate for the required execution time, but presupposes considerable knowledge of numerical techniques on the part of the programmer. This presupposition defeats in large part the basic idea, i.e., the simplicity and ease of use, even for relatively untrained people.

In sum, it appears at this time that the debate is hot on integration methods, and more experience is still required. Parenthetically, it might be said that an objective, thorough comparison of the various options would be a real service to the field of digital simulation.

2. **One Frame Delay:** This element delays its input by one integration interval. It should be available for the sophisticated user to selectively “desort” the program list. COBLOC and DSL/90 presently have the option of sort/no sort, but if the no sort option is required in only one small area, much care must be taken in the other sections to insure proper operation. The one frame delay is also quite useful for representing sampled data systems or memory functions.
3. **Hysteresis or Backlash**: This element is not easily constructed from standard analog type elements, but represents a trivial task for the digital computer.

4. **Limited Integrator**: Again, no easy chore from the standard elements, and no real effort for the digital computer. MIMIC presently has a limited integrator element which is used in conjunction with a standard integrator.

5. **Transfer Functions**: These operators are used extensively in control system design and the like, and are simply constructed from analog type elements. However, the very frequency of their use suggests they be made available as integrated general-purpose units.

   In addition to the programming time savings, execution time can be saved, since the integration algorithm required for a closed loop transfer function is much simpler than a comparably accurate routine for open loop integration.

   Far greater savings are possible by using a difference equation algorithm. This method requires only one computational step of the same size and complexity of a single integration. Compare this with the n integrations required for a transfer function with nth order denominator, when programmed by normal analog methods.

6. **Print Operators**: Very often, in checkout and operation, it would be helpful to force a print (number or words) at an arbitrary point in the program. The operator would be similar to a “snapshot” print, but would be under the control of problem variables. As a trivial example, consider the printing of “OVERLOAD” when an analog check case variable exceeds 100.

7. **Parallel Logic**: The operators of interest here are the normal digital units found on most modern analog computers. (AND gates, flip-flops, counters, shift registers, etc.) These elements are presently available in the MADBLOC, COBLOC, and MIMIC languages. Their inclusion is essential to provide a digital check for a modern analog computer problem. When solving differential equations, such units are also useful for simple logic and storage.

   For the higher level logic of the type normally associated with general-purpose digital computers, machine language subprograms, as discussed above, are more useful.

8. **Linkage Elements**: For hybrid programming, elements or labels for analog to digital converter (ADC) and digital to analog converter (DAC) components must be provided. ADC's could be handled simply as another parameter input to the problem. These could be realistically labeled and then identified somewhere in the program listing, e.g., ALPHA = ADC1. DAC's could also be easily handled; for the digital program they are merely elements with one input and no output. Sorting these elements presents no difficulties: ADC's would be treated exactly like constants; DAC's would be sorted like any other element which has an input.

9. **Hybrid Integrators**: Since variables transferred to the analog computer are usually held for an entire frame, an effective half interval time lag results. Thus, integrators generating quantities destined for the analog must account for this lag phenomenon. Those integrators in a purely digital loop must, of course, neglect this extrapolation. Therefore, different integrator types, easily distinguishable and easily specified, must be provided for the two requirements. It is entirely possible that the sorting routine could automatically make the necessary distinctions by tracing back from a DAC to integrators not isolated by another integrator. An alternate procedure is the addition of an extrapolation calculation to each DAC element. However, this approach costs both storage and execution time.

**CONCLUSIONS**

Digital simulation languages have made a real, and probably permanent, impact on the fields of both simulation and computer programming. As has been pointed out, there are more or less serious faults in all existing languages. The success of the approach is, however, evidenced by the undeniable acceptance, utilization and enthusiasm for simulation languages, regardless of the difficulties at the present phase of development.
It is the authors' hope that the conclusions and recommendations proposed herein will add significantly to the utility of simulation languages, and the field will enjoy even further growth and acceptance.

REFERENCES


