Reprinted from:
COMPUTER SCIENCE AND SCIENTIFIC COMPUTING
© 1976
ACADEMIC PRESS, INC.
New York San Francisco London

LANGUAGE AS A TOOL FOR SCIENTIFIC PROGRAMMING

Victor R. Basili
Computer Science Department
University of Maryland

ABSTRACT

Programming languages act as software development tools for problems from a specific application area. The needs of the various scientific programming applications vary greatly with the size and style of the problems which include everything from small numerical algorithms to large-scale systems. This latter requires language primitives for a multitude of subproblems that include the management of data, the interfaces with the system at various levels, etc. One way of making available all the necessary primitives is to develop one very large language to cover all the needs. An alternative is to use a hierarchical family of languages, each covering a different aspect of the larger problem, e.g., a mathematical language, a data base management language, a graphics language, etc. The concept of a family of languages built from a small common base offers a modular, well-interfaced set of small specialized languages that can support such software development characteristics as modularity, reliability, efficiency, and transportability. This paper discusses the use of the hierarchical family concept in the development of scientific programming systems.

I. INTRODUCTION

Tools are designed to aid in the construction of a product. They help produce a better product and make the actual building process easier. They help in the automation of the various stages of development making the more tedious, difficult, and error-prone aspects easier and more reliable.

Scientific programming leads to the development of products. What do these products look like? They vary in size and complexity from library routines for basic mathematical functions to self-contained programs encompassing basic algorithms to large

whole systems involving structural analysis (NASTRAN [NAS72], NONSAP [BAT74], . . .) or control center operations (MSOCC [NAS75], ATSOCC [DES75], . . .).

Tools have been developed to aid the scientific programmer in product development. Consider the problem from an historical perspective. In the beginning there was the bare machine. The solution to the scientific programming problem was expressed as a set of machine language instructions. The programmer developed everything from scratch every time, working only with the basic hardware. Almost immediately tools were developed to help specify programs in more human-related forms improving the machine environment in which the programmer worked.

The major emphasis in tools for expressing the problem solution has been in the development of higher-level programming languages. Such languages create an idealized, virtual computer that corresponds to the thinking habits and capabilities of the programmer, rather than to the limitations of the current technology.

One of the first tools developed was the library concept permitting the reuse of existing subprograms. In this way common tasks, such as mathematical functions, could be programmed once and used over and over again. These subprograms were a first step in defining a higher level set of the programming primitives suited to the user's needs and the application area.

A second tool was the development of symbolic codes. Assembly languages permitted the user to write in a symbolic notation to specify the instructions and locations of the machine. The use of mnemonics for instruction primitives made it easier for the programmer to relate to the data of the program. This higher level of specification was made possible by the assembler which translates the symbolic codes into the real machine codes.

Higher-level scientifically-oriented languages, such as FORTRAN and ALGOL, were developed to automate more powerful

forms of specification. They incorporated specifications for arithmetic expressions, data such as complex variables, control structures such as conditionals and iteration, and data structures such as arrays. These higher-level languages are translated using compilers or macroprocessors or executed via interpreters.

A set of high-level, more general-purpose languages, such as PL/1 and HALS/S [NEW73], offered facilities to the programmer not available in languages such as FORTRAN. These facilities included string processing and some control over the system environment in the form of overflow and underflow control.

In order to more closely associate the terminology for expressing the problem solution with the problem area, special purpose application oriented languages have been developed. Languages such as SNOBOL and LISP were developed early to handle string and list processing problems, respectively. There has been an ever-increasing number of very high-level languages that involve the solutions of set-theoretic problems (e.g., SETL [SCH73]), combinatorial problems (e.g., MADCAP [WEL70]), artificial intelligence based problems (e.g., SAIL [FEL72], PLANNER [SUS70]), graph algorithmic problems (e.g., GRAAL [RHE72], GEA [CRE70]), etc. Some of these languages have been called very high-level languages as they raise the level of specification far above that defined by the high-level languages, e.g., a primitive like the union operator in SETL would require a subroutine in FORTRAN.

Besides being used to raise the level of the machine to help the programmer express his program in the language of the problem area, tools have been developed to improve the environment in which the programs and programmer live. On top of the bare machine, monitors were developed to run the computer without stopping, automatically sequencing batches of jobs. This helped make more effective use of machine time and removed the programmer from pushing buttons on the bare machine. Under such a

system, however, interactive development was lost. In time, sequential batch processing gave way to demand systems. Using a multiple programming system with a virtual memory, each user was effectively given his own machine, in spite of the fact that he was one of many users. These systems gave the user a greater opportunity to share high-level resources such as text editors and file systems. They returned to the programmer the ability to interact with the machine, but this new software-extended machine was more powerful and several levels closer to man's problem areas.

Support tools have been developed which try to provide the programmer with an environment for developing and analyzing his programs at the level at which they are written. These support tools include debugging aids, testing and evaluation aids, and a variety of support programs including editors and data base analyzers.

Given this large assortment of tools, the programmer needs a mechanism for harnessing all these capabilities. Consider the definition of a programming language in this context. *A programming language is a standardized notation used to express a problem solution and communicate that solution to humans and computers.* (Note that the idea of expressing that solution to humans has gained considerable importance in light of the problems of correctness and maintainability.) The programming language must provide the user with control over whatever primitives are needed to express the problem solution and implement that solution as a correctly executing program. All of the tools mentioned earlier can be harnassed using programming language notation.

The definition given here emphasizes several important points which should characterize a programming language. These include ease of expression, the ability to write correct, readable, implementable and efficient programs. The ability to communicate with various computers implies portability. Another

important characteristic of a programming language is the ability to reuse products developed in the language.

The next section discusses some of the capabilities and characteristics a scientific programming language should have. Section 3 recommends a method for defining a programming language notation that achieves this set of capabilities and characteristics using the concept of a family of languages.

II. SCIENTIFIC PROGRAMMING LANGUAGE NEEDS

The scientific programming notation must provide the programmer with a set of capabilities for solving his problem and this notation should have characteristics that support the ease of expression, correctness, etc., of the solution algorithm. First, consider some of the actual capabilities a scientific program via notation might encompass.

The most basic facility that is common across product size and complexity is the ability to perform numerical calculations. The notation should provide a basic set of arithmetic data types including integer, real, complex. For operation on these basic data types there should be a complete set of built-in operators for each data type, along with complete libraries of special functions. There should be clearly defined hierarchies of conversions, e.g., integer is a subset of real and real is a subset of complex and mixed types are automatically converted to the higher level type. The user should also have control over overflow and underflow with respect to the various operations.

There should be a variety of precisions for data types; for example, there should be short and long reals, short integers, etc. The user should have some control over precision in that it should be easy to change precision across an entire program. One way the need for a specific precision can be satisfied is by a variable precision arithmetic package. This can be very inefficient however because the specified precision may not correspond to the machine word size boundaries. Specific precision requirements could be satisfied more efficiently by permitting the

user to specify a minimum precision requirement and permitting the compiler to impose a precision greater than the specified precision that corresponds to some multiple of the machine word size [INT75]. Precision hierarchies should be clearly defined just as data type hierarchies. All functions should appear to the user to be generic with respect to precision and data type. There should be no need for the user to remember different function names for different data types and precisions.

The notation should provide a convenient format for arithmetic expressions. Since random reordering of expression evaluation can cause problems with precision critical computations, the evaluation order of specific expressions must be preserved. However, the user is also interested in optimization of non-precision-critical expressions, so there should be some control conventions or special formats to forbid reordering. For example, the compiler might assume that normal parenthesized expressions, or some special form of parenthesized expressions, cannot be reordered.

What about the framework in which one imbeds arithmetic expressions? This includes the control structures, data structures, and the runtime environment. There has been a great deal in the literature about good sets of control structures for writing algorithms [DAH72,MIL75]. These include the standard sequencing, the ifthenelse, whiledo, etc. These structured control structures aid in the development of readable, correct algorithms. Of special interest in scientific computations are an assignment mechanism and the indexed loop statement, i.e., the for or do loop. There are two formats for assignment. One is to consider assignment as an operator, as in APL [PAK72]; the second is to consider it as a statement as in FORTRAN. Studies [GAN75] have shown that the assignment statement as opposed to the assignment operator appears to be a less error prone construct. The main benefits of the indexed loop statement are as an aid to correctness (automatic indexing built in) and for efficiency (the loop

variable can be specially treated in an optimal way if the use of the index variable is limited to indexing, i.e., it is not available outside the loop and it cannot be assigned a value inside the loop.)

With respect to data structures the standard workhorse has been the array. However for many situations what is needed are different methods of access. For example, one may want to access an array of data both as a one-dimensional array and as a two-dimensional array depending upon whether speed or ease of access is of interest. Interestingly enough, FORTRAN allows the user this kind of control over its array data structure. It permits the programmer to use the array as a storage map over which different templates may be defined. Whether this facility was an accident of design or purposeful, it is a widely used feature of the language and gives the user a great deal of power with respect to data structuring. It is this kind of mechanism that language designers are trying to improve upon and build into languages in a less error prone way [LIS74,WUL76].

The runtime environment involves the language framework in which the control and data structures are imbedded. For example, the procedure organization (internal procedures, external procedures, nested procedures), the data scoping (block structure, common blocks, equivalencing,...) and the existence of special facilities (recursion, pointer variables, dynamically allocated storage,...) all contribute to the complexity of the runtime environment. From the points of view of efficiency and ease of understanding it is important to keep the runtime environment as simple as possible. It is worth noting that FORTRAN has a relatively simple and efficient runtime environment compared to a language like PL/1, which may account for part of its strong support by the scientific programming community. A simple runtime environment framework can be a real asset in a programming language.

The language facilities discussed so far have been a variation and extension of the facilities found in languages like FORTRAN and ALGOL. However, many of the needs of scientific programming go beyond the facilities mentioned so far. For example, string and character processing are needed for formatting outputs, generating reports and processing data in nonnumeric form. Besides the string or character data types, basic operators are needed along with conversion routines for changing between numeric and nonnumeric data.

When the size of the problem and the amount of data become large, the ability to interact with the system environment becomes necessary. The facility to read and write files is needed for storing and retrieving large amounts of data. In order to fit various segments of the program and data into memory at one time, the user needs access to overlay capabilities. Larger size implies more control over input and output routines and devices. And, of course, as mentioned earlier, control over interrupts plays an important role in handling overflow and underflow, etc.

As the problem gets more specialized, new features may become appropriate. Thus, the scientific programming notation might provide some extensibility capabilities for creating new features as necessary. For example, a matrix data type might be a convenient primitive, along with a set of matrix operations. It would be convenient for the programmer to be able to build this into the notation easily to make expressing his problem easier.

Thus far in the discussion we have tried to motivate a set of capabilities building bottom up starting with existing capabilities in standard scientific programming languages, and adding some of the facilities that are needed as the problem gets larger or the application becomes more specialized. Let us view the situation from the top down with respect to the needs of large-scale scientific programming applications. These large-

scale applications involve the development of entire systems. Consider examples such as structural mechanics systems, e.g., NASTRAN, NONSAP, and control center systems, e.g., MSOCC, ATSOCC. Are these scientific or system applications? If they are examined from the user's point of view, they are definitely scientific applications.

What are these large scientific programming systems composed of? That varies with the particular application. However, there are some aspects that many of them have in common. At the very top is the user's interface to the system which is some form of system operation language (SOL). The power of the SOL is dependent upon the flexibility of the system and the sophistication of the user. SOLs vary from a set of push buttons to some limited form of a data input to some minimal form of sequential control to a full high-level language. Ideally there should be some flexibility in the level of the language. The unsophisticated user should be able to enter the system at a very high level using some standard prepared package routines. The more sophisticated user should be given more flexibility and be able to penetrate the system at the level of his expertise. Thus, the SOL should be both flexible and hierarchical.

To demonstrate the scope a programming notation must address for a large-scale scientific system, consider a particular application area, such as structural mechanics. What are the various aspects of such a system?

The user aspect involves an engineer who wants to check out some structure. At the top level he would like some simple form of data input for using a canned set of techniques. However, if the user is reasonably sophisticated and the canned techniques don't adapt well to the particular problem, he should be allowed to write a high-level algorithm to bring the power of the system to bear for his particular needs. The numerical analysis aspect involves many numerical algorithms for the solution of differential equations, non linear equations, eigenvalue extraction, etc.

There is a data base management aspect. This involves the definition of data structures for the structural elements, the definition of the grid point storage structure, the physical representation of these structures, and the access and storage techniques. There is a graphics aspect. The user should be able to display the input graphically for checking and the output for interpreting results easily. Finally, there is the systems aspect. One has to build a system in which all of the above processing takes place. This system should in fact include some interactive facility.

The above discussion is clearly superficial. It is merely meant to demonstrate the wide variety of facilities required in the notation for building large-scale scientific systems.

Having all of the above capabilities, the scientific programming notation should possess certain characteristics. Among other things, it should support ease of problem expression, the writing of correct, efficient, and portable code, and the reuse of algorithms written in it. Let us consider these characteristics one at a time.

One would like to express algorithms in a natural manner. This implies the notation should be natural to the problem area. For example, within the general problem area of mathematics there is a specialized and different mathematical notation for the algebraist and analyst. Each aid in expressing the problems of the particular area explicitly and precisely and in an easy to communicate form.

However, it is hard to define the right features for the application area. Often knowing just the right notation is part of the solution. We need to experiment with language features. The notations of algebra and analysis have been refined over many years. To work on this problem of defining the right notation for a particular application, one needs both a language expert and an application area expert. Only the application area expert can know the right abstractions for the problem area. But the language expert is needed to model and

analyze these abstractions for ease and efficiency of implementation, error proneness, and interaction with other facilities in the language and the environment. We have learned a lot about language design in the last two decades and we should use this knowledge in the design of languages. For example, all new languages should be modeled using several modeling techniques [BAS75a,HOA73,HOA74,LUC68] to guarantee good design with respect to the criteria that each of the models demonstrates.

Correctness of a program is defined as the ability of the program to perform consistently with what we perceive to be its functional specifications. The programming language should support the writing of correct programs. The language should simplify rather than complicate the understanding of the problem solution. The complexity in understanding a program should be due to the complexity inherent in the algorithms, not due to the notation used. The notation should be clear and simple. A language natural to the problem area aids in correctness as it makes the statement of the solution easier to read and understand. The easier it is to read and understand a solution algorithm, the easier it is to certify its correctness. Aids in making a program readable are to structure it from top to bottom and to break it into small pieces. In order to achieve the goal of supporting correctness, a language should be simple, contain well-structured control and data structures, permit the breaking up of the algorithm into small pieces using procedures and macros, and contain high-level problem area oriented language primitives.

A program is considered efficient if it executes at as fast a speed and in as small a space as is necessary. The language should permit the efficient execution of programs written in it. The higher level the algorithm, the more information is exposed for optimization and the better job a compiler can do on improving the code generated. On the other hand, high level often implies general applicability in order to handle the majority of cases. This can often imply an inefficiency for a particular

application. For example, consider a language in which matrices have been defined as a primitive data type with a full set of operators including matrix multiplication. The multiplication operation has been defined for the general case. Suppose the particular subproblem calls for the multiplication of two triangular matrices. Using the standard built-in operator is inefficient. One would like to be able to substitute a more efficient multiplication algorithm for the particular case involved. But this implies that the language permits the redefinition of language primitives at lower levels of abstraction. That is, the programmer should be able to express the algorithm at a high level and then alter the lower level design of the algorithm primitives for a particular application when it is necessary for reasons of efficiency.

A language supports portability when it permits the writing of algorithms that can execute on different machines. Portability is a difficult, subtle problem that involves several diverse subproblems. The numerical accuracy of arithmetic computations can vary even on machines with the same word size. Techniques for dealing with this problem include variable length arithmetic packages or a minimum precision (modulo word size) specifications as mentioned earlier. Another problem area of portability is text processing. One way of dealing with this problem is to define a high-level string data type which is word size independent. A third area of problems involves interfacing with a variety of host machine systems. One method of handling this is to define programs to run on some level of virtual machine that is acceptable across the various machine architectures and systems and then to define that virtual machine on top of the host system for each of those architectures. This is commonly done using a runtime library. In general the higher level the algorithms, the more portable they are. However, more portability often means less efficiency. A language that supports portability should contain one of the above mechanisms

for transporting numerical precision across machine architectures, high-level data types, the ability to keep nonportable aspects in one place, and a macro facility for parameterizing packets of information modulo word size.

Software is reusable if it can be used across several different projects with similar benefits. In order for software to be reusable, its function must be of a reasonably general nature, e.g., the square root and sine functions, it must be written in a general way and it must have a good, simple, straightforward set of specifications. The area of scientific programming has a better history of reusable software than most. Consider as examples some of the libraries of numerical analysis routines [INTE75,SMI75]. This is due largely to the easily recognizable, general nature of many scientific functions and the simplicity of their specifications. There are whole areas of scientific software development, however, that do not have a history of reuse. Consider telemetry software, for example.

Software written in a general way may perform less efficiently than hand-tailored software. However, if it is well written it should be possible to measure it and based on these measures modify it slightly in the appropraite places to perform to specification for the particular application.

A good, simple, straightforward set of specifications is not easy to accomplish, especially when the nature of the function is complex. A good high level algorithm can help in eliciting that specification. Specifications for software modules should also include an analysis of the algorithm, e.g., the efficiency of the algorithm with respect to the size of the input data. The language should support the development of a good library of well-specified software modules that are easy to modify if the time and space requirements are off. It should also be capable of interfacing efficiently with other languages and of expressing

algorithms so that the essential function is clear and of a general nature.

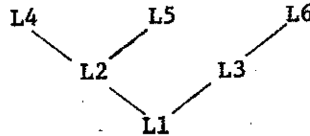
This partial list of capabilities and characteristics for a scientific programming language contains a large number of seemingly contradictory requirements. It is quite possible we can't have a notation with all of the above characteristics. However, one would like to define a language tool in which we can maximize both the capabilities and the above characteristics.

III. A FAMILY OF PROGRAMMING LANGUAGES

To begin with, one language appears to be not enough. If we are to cover all the applications necessary, the language would be too large and contain too many contradictory features. The runtime environment needed to support such a language would be complex and inefficient. What we would like is a set of languages, each tailored to a particular subapplication. However, there are several drawbacks to building a large set of independent languages each tailored to a subapplication. For one thing, the design and development of new programming languages are fraught with many problems since each language would be an entirely new design experience. Secondly, if these languages were truly different in design, it would require the user to learn several totally different notations for solving the different aspects of the problem. Thirdly, there would be a proliferation of languages and compilers to maintain.

One possible approach that minimizes some of the above drawbacks is the development of a family of programming languages and compilers. The basic idea behind the family is that all the languages in the family contain a core design which consists of a minimal set of common language features and a simple common runtime environment. This core design defines the base language for which all other languages in the family are extensions. This also guarantees a basic common design for the compilers. The basic family concept can be viewed as a tree structure of languages in which each of the languages in a subtree is an

extension of the language at the root of the subtree, i.e.,



and

$$L4 = L2 \cup \{\text{new features of } L4\}.$$

Using the family approach permits the development of several application area languages, minimizing the difference between the languages and the compiler design effort. Since many of the constructs for various applications contain a similarity of design or interact with the environment in similar ways, experience derived from one design and development effort can be directly applicable to another. Since the notation for a particular application area may not be totally clear a priori, the family idea permits some experimentation without the cost of a totally new language and compiler development.

There are several approaches to minimizing the compiler development for a family of languages. One can develop an extensible language and build the family out of the extensible base language. The extension can be made either by a data definitional facility as in CLU [LIS74] or ALPHARD [WUL76] or by some form of full language extension as in ELF [CHE68] or IMP [IRO70]. The family of compilers can be built using a translator writing system [FEL68] or by extending some base core compiler [BAS75b]. A combination of two of the above techniques is recommended here, and they will be discussed a little more fully.

In the core extensible compiler approach, the base compiler for the base language is extended for each new language in the family, creating a family of compilers, each built out of the base core compiler. In order to achieve the resulting family of compilers, the core compiler must be easy to modify and easy to extend with new features. One experience with this technique, the SIMPL

family of languages and compilers [BAS76] has proved reasonably successful with respect to extensibility by using specialized software development techniques to develop the compiler [BAS75c].

Using the core extensible compiler approach, the compiler C(L) for a new language in the family is built from the compiler at its father node on the tree. This is done by making modifications (mod) to that compiler to adjust it to handle the new features of the extension, i.e.,

$$C(L4) = C(L2)\underline{mod}\{L4\text{-fixes}\} \cup \{L4\text{-routines}\}$$

where the set of L4 routines represent the code for the L4 extensions to L2 and the set of L4 fixes represent the code for modifying the L2 compiler to add those extensions. The key to good extensible compiler design is to minimize the number of modifications (fixes) and maximize the number of independent routines.

Using a data extension facility, new data types and data structures can be added to the language using a built-in data definition facility. In order to achieve reasonable extensibility, the facility should be easy to use and permit efficient implementation. Experience with forms of data abstraction facilities in CLU, ALPHARD, and Concurrent PASCAL [HAN75] have deomonstrated the benefits of this approach.

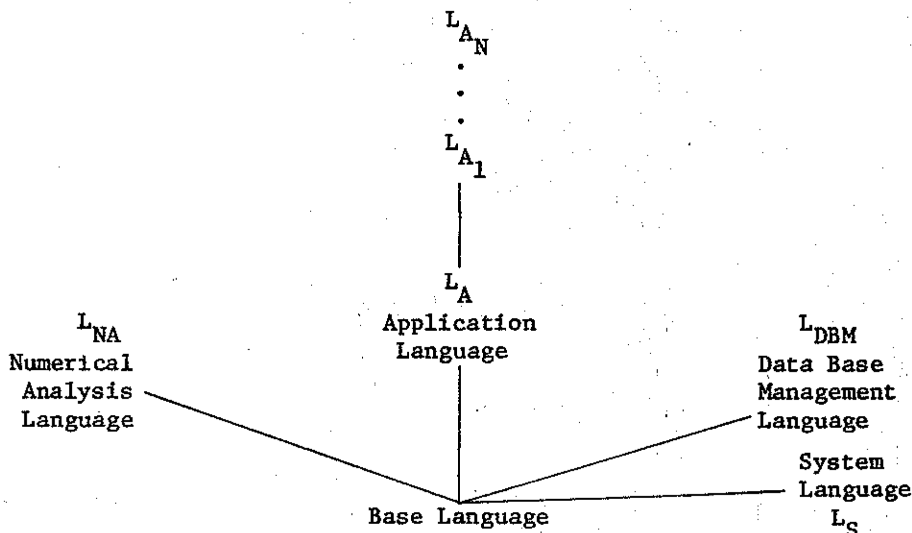
Here the effective compiler for the new language is again built from the compiler at its father node on the tree. This is done by adding a new set of library modules that represent the new data types and structures and associated operators and access mechanisms, respectively, of the new language, i.e.,

$$C(L4) = C(L2) \cup \{L4\text{-library}\}.$$

Each of the two techniques has different assets. The core extensible compiler approach permits full language extension, including new control structures and modifications to the runtime environment. It offers the most efficiency and permits a full set of specialized error diagnostics to be built in. The data definitional facility can be used only for data extensions,

but these are by far the most common in the range of subapplications. It is also a lot easier to do and can be performed by the average programmer, where the compiler extensions require more specialized training. Ideally, the first approach should be used for major application extensions and the second for smaller subapplication extensions.

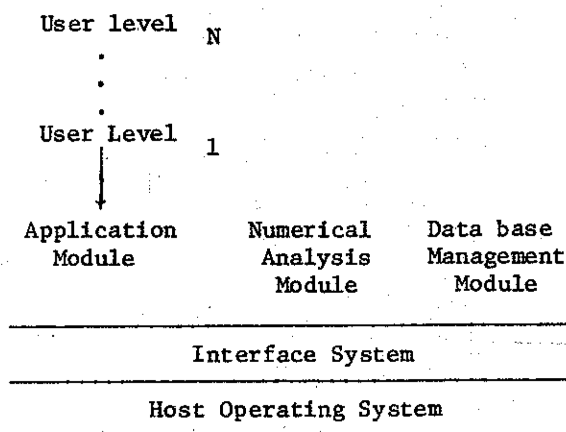
Let us now apply this family concept to a large-scale system concept and analyze how the various application-oriented language features could be distributed across several languages in the family. Consider the structural mechanics system discussed earlier. There would be a language in the family for each application, i.e., an engineering application language, a numerical analysis language, a data base management language, a graphics language, etc. Each language would be built out of some base language (which may in fact be the system language). The application language may have several extensions, each of which adds on some higher level set of primitives, e.g., some set of standardized algorithms could be defined as a simple set of primitives. The family tree for the languages may take on a form such as



The system is then modularized so that each module is programmed in the appropriate language, i.e., the numerical analysis module in the numerical analysis language L_{NA} , etc. The application area module is programmed in the application area language L_A . However, the SOL can be just as high an extension of L_A as is appropriate for the sophistication of the user.

Each of these modules interfaces with one another through an interfacing system. The interface system is part of the basis for the family of languages and contains among other things the compilers for the languages. The interface system is an extension to the host operating system for whatever computer the structured mechanics system is being developed. Together with the host operating system, it makes up the virtual machine in which the modules interact.

This kind of system offers a variety of SOLs. The unsophisticated user can interface into the system at the top level L_{A_N} . The more sophisticated user can drop down any number of levels to gain more power over the system by using lower level languages in the hierarchy.



It is clear that the family of languages concept permits the incorporation of the various capabilities required for a large-scale scientific programming system. How does this concept rate with regard to the criteria discussed earlier, i.e., ease of

expression, correctness, efficiency, portability and reusability?

With resepct to ease of expression, algorithms are written in a notation which is specialized to the application. In fact, since each language is reasonably independent of the application level, and primitives in one notation can be fine-tuned without affecting the primitives of another application, this permits a certain amount of experimentation, and primitives can be varied with experience.

High-level, application-oriented primitives make a solution algorithm easier to read and understand and therefore easier to prove correct. The specialized notation raises the level of the executing algorithm to the level at which the solution is developed. Individualized compilers tailor error diagnostics and recovery to the particular application.

Each language is small and relatively simple. Compilation of programs is very efficient, and recompilation of primitive definitions even at runtime is reasonably efficient. Each language is not complicated by a mix of features whose interaction may complicate the runtime environment. A more tailored runtime environment implies more efficient execution at runtime. Language features are specialized to meet the application and don't have to be generalized, inefficient versions of the feature. Because of the hierarchical structure of the languages, the programmer can use lower level languages to improve or fine-tune algorithms when necessary.

Higher level primitives mean more portable algorithms. The hierarchy with respect to the data abstractions permits the localization of the nonportable aspects of the program that can be recoded for a new machine architecture in a lower-level language in the hierarchy.

With regard to the development of reusable software, each application area has its own language. Thus, needed submodules are written in the target application notation rather than the host application notation. This makes it easier to recognize

the essential function of the submodule and easier to write it in a more generally applicable way.

Several general comments should be made about the family approach. Since the languages are all members of a family, it is relatively easy to move from one language to another because of the common base. Whole new notations do not need to be learned. Interfacing with different languages is easier since there is a common base language/compiler system. However, good interfacing must be designed in from the start. The high-level primitives can remain fixed and be redefined at lower levels for purposes of portability or efficiency. There are several compilers to maintain although one approach might be to store one source in macro with expansions for each of the languages. It is harder to build an easily extendable/modifiable compiler and requires more care in development. But ease of modification should be an important property of any software. Some inefficencies exist with respect to the data abstractions passed between modules. However, these problems are being studied and the use of an interface system may help a great deal in effecting an efficient mechanism.

IV. SUMMARY

Scientific programming involves a multi-facted set of needs which vary from small programs to large systems among several subapplication areas. A scientific programming notation is needed for solving these problems in an effective way where effective means easy to express, correct, efficient, portable, and reusable. One way to conquer the complexity of the problem is to use as a notation a special application-oriented family of languages, one language for each subproblem area, and interface these languages into a portable system that can be built on top of a standard operating system.

REFERENCES

[BAS75a] Basili, Victor R. A Structured Approach to Language Design. *Journal of Computer Languages, Vol 1,* pp. 255-273 (1975).

[BAS75b] Basili, V.R. and Turner, J. Computer Science Center, University of Maryland, Technical Report #269, 14 pages. A transportable extendable compiler. *Software Practice and Experience, Vol. 5,* pp. 269-278 (1975).

[BAS75c] Basili, V. R. and Turner, A. J. Iterative Enhancement: A Practical Technique for Software Development. Proceedings of the First National Conference on Software Engineering, Washington, D.C., September 11-12, 1975.

[BAS76] Basili, Victor R. The SIMPL Family of Programming Languages and Compilers. Graphen-Sprachen und Algorithmen auf Graphen, Carl Hansen Verlag, Munich, Germany, 1976, pp. 49-85. Also Computer Science Technical Report #305, University of Maryland, June, 1974.

[BAT74] Bathe, Klaus-Jürgen, Wilson, Edwarld L. and Iding, Robert H. NONSAP A Structured Analysis Program for Static and Dynamic Response of Nonlinear Systems. Dept. of Civil Engineering, University of California, Berkeley, Report No. UCSESM74-3, February 1974.

[CHE68] Cheatham, T. E., Jr., Fischer, A. and Jorrand, P. On the basis for ELF - an extensible language facility. Proc. AFIPS 1968 Fall Joint Comput. Conf., Vol. 33, Pt2, pp. 937-948. The Thompson Book Co., Washington, D.C.

[CRE70] Crespi-Regluzzi, S. and Morpurgo, A language for treating graphs. *COMM. ACM,* pp. 319-323, May, 1970.

[DAH72] Dahl, O. -J., E. W. Dijkstra, and C. A. R. Hoare. Structured Programming. Academic Press, 1972.

[DES75] desJardins, R. and Hahn, J. ATS-6 Control Center Real-time Graphics Displays. Digest of Papers, IEEE COMPCON 75 Fall, Sept. 9-11, 1975, pp. 234-6.

[FEL68] Feldman, J. and Gries, D. Translator Writing Systems. *COMM. ACM, 13, 2,* (Feb. 1968) pp. 77-113.

[FEL72] Feldman, J. A., et al. Recent developments in SAIL-An ALGOL based language for artificial intelligence. Proceedings FJCC, AFIPS Press, Vol. 41 Part 2, Montvale, N.J. (1972) pp. 1193-1202.

[GAN75] Gannon, John and Horning, James. The Impact of Language Design on the Production of Reliable Software. Proceedings of the Interantional Conference on Reliable Software, April 1975, pp. 10-22.

[HAN75] Hansen, Per Brinch. The Purpose of Concurrent PASCAL. Proceedings of the International Conference on Reliable Software, April 1975, pp. 305-309.

[HOA73] Hoare, C. A. R. and Wirth, N. An Axiomatic Definition of the Programming Language PASCAL. *Acta Informatica 2,* pp. 335-355, 1973.

[HOA74] Hoare, C. A. R. and Lauer, P. E. Consistent and Complementry Formal Theories of the Semantics of Programming Languages. *Acta Informatica, Vol. 3, fasc: 2,* 1974, pp. 135-153, Springer-Verlag-Berlin.

[INT75] CS-4 Language Reference Manual and CS-4 Operating System Interface. Intermetrics, Inc., Cambridge, Mass. October, 1975.

[INTE75] IMSL Library 2 Reference Manual, International Mathematical and Statistical Libraries, INC., Houston, Texas, 1975.

[IRO70] Irons, E. T. "Experience with an Extensible Language". *COMM. ACM, 13, 1* (Jan. 1970), pp. 31-40.

[LIS74] Liskov, B. and Zilles, S. Programming with Abstract Data Types. Proceedings of a Symposium on Very High Level Languages, March, 1974, SIGPLAN Notices, Vol. 9,

No. 4 April 1974.

[LUC68] Lucas, P. Lauer, P., and Stiglertner, H. Method and notation for formal definition of programming languages. IBM Tech. Rep. 25-087, IBM Lab., Vienna, 1968, 107 pages.

[MIL75] Mills, H. D. How to Write Correct Programs and Know It. Proceedings of the International Conference on Reliable Software. Los Angeles, California, April, 1975, pp. 363-370.

[NAS72] NASTRAN User's Manual. NASA SP-222(01). NASA Scientific and Technical Information Office, Washington, D. C., June 1972.

[NAS75] Multi-Satellite Operations Control Center (MSOCC) Systems Manual. NASA/Goddard Space Flight Center, August 1975.

[NEW73] Newbold, P. M. and Helmers, C. T., Jr. HAL/S Language Specification. Intermetics, Inc. Cambridge, Mass. April 1973.

[PAK72] Pakin, Sandra. APL\360 Reference Manual. SRA, 1972.

[RHE72] Rheinboldt, W. C., Basili, V. R. and Mesytenyi, C. On a Programming Language for graph algorithms. *BIT 12* (1972) pp. 220-241.

[SCH73] Schwartz, J. On Programming: An Interim Report on the SETL Project, Installment I, II, 1973. Also Computer Science Department Courant Institute of Mathematical Science, New York University.

[SMI74] Smith, B. T., Boyle, J. M., Garbow, B. S., Ikebe, Y., Klema, V. C., Moler, C. B. Lecture Notes in Computer Science V6: Matrix Eigensystem Routines - EISPACK Guide. New York: Springer 1974.

[SUS70] Sussman, G. J. etal. MICRO-PLANNER Reference Manual MIT Artif. Intel. Lab., Memo No. 57.1 (April 1970).

[WEL70] Wells, M. B. Elements of Combinatorial Computing. Pergamon, Oxford, 1970.

[WUL76] Wulf, Wm. A., London, Ralph, L., Shaw, Mary. Abstraction and Verification in ALPHARD. February 1976, (unpublished memorandum). Department of Computer Science, Carnegie Mellon University.