METACYBERNETICS

Design Evolution of Metacomputer Host Architecture

by

Joseph Thames

The Aerospace Corporation

ABSTRACT

Metacybernetics or metacomputer cybernetic development is a technology of cybernetic design facilitated by an "evolving host" architecture. It is an outgrowth of 2nd generation software engineering, focusing on meta design -- language-directed design of language emulation architecture. Applications are produced using adaptive high-order languages that are executed by metacomputers -- ad hoc computing engines in host micro-software that interpret assembled very-high-level token streams (programs). These computing engines are themselves implemented and evolved by metalanguage directed generators resembling compiler-compilers.

The motivation is to provide a medium of design evolution, driven by application feedback, which can become a vehicle for knowledge acquisition supporting with the development of expert systems, robots, and autonomous control. The goal is the direct adaptation and evolution of the application interfaces of computer architecture by programmers and engineers who directly support applications.

CONTENTS

INTRODUCTION
1.0 A STRATEGY OF ADAPTIVE ENGINEERING
2.0 ORIGINS AND CONCEPTS OF META DESIGN
- Metalanguages and Metaphoric Application Languages
- Architecture Requirements

3.0 METACOMPUTER DESIGN SYNTHESIS
- Metacomputer Form and Genesis
- Metacomputer Evolution

4.0 METACYBERNETIC ENGINEERING
- Design Evolution Mobility
- Performance Engineering
- Metacomputer Host Environment

5.0 CONCLUSION

INTRODUCTION

The Aerospace Research Computer (ARC, Ref. 1) is the prototype foundation for an evolving language-directed technology of cybernetic design. This technology generalizes the essential character of very-high-order programming, i.e., the automatic generation of complete detailed designs from abstract descriptions, to encompass the full spectrum of cybernetic logic, including software, firmware and hardware. It is a technology of meta-specified* design automation, or meta design, deriving from compiler-compiler technology.

The departure from historical computer architecture necessary to facilitate meta design is not a radical change in technology. It is rather a realignment of existing technology to fit a development paradigm that has proven itself in software R & D, but is heavily penalized by historical architectures. The approach is a re-application of M.V. Wilkes' original concept of microprogramming to implement metacomputers as high-order software platforms. The supporting hardware architecture therefore becomes a metacomputer host.

A metacomputer is a programmed adaptation of the metacomputer host. It is a total computing environment consisting of a micro-programmed interpreter, its associated instruction-set architecture and the programming language that it executes. A metacomputer host may have many of these environments, addressing a variety of application classes.

In the context of historical microprogrammable mainframes, most computers (real or virtual) are low-level dedicated metacomputers. Moreover, they are frozen to the historical von Neumann paradigm of memory-oriented software. They emulate historical machine-code images which are far below the optimum level of programming language. High-order metacomputers, unlike historical machines, originate from software emulation of high-order languages. Consequently, they need not adhere to the historical memory-addressing paradigm nor to the primitive level of machine image programming. Their instructions may be free-form token strings devoid of binding hardware addresses. Alternatively, metacomputers may be knowledge-based expert-system interpreters, whose instruction streams are retrieved from data bases having no physical bounds or residence. The extent of programs and data are unlimited in scope, as this is the inherent nature of high-level language.

---

* "meta-specified" refers to an indirect means of design description in which the description is a highly abstracted "metaphor" of the actual design structure that is produced.

The motivation for a technology of meta design is to shift the burden of design diversity from a centralized, unified technological kernel of computer science (the computer manufacturer) to a highly decentralized, diversified domain of application science and engineering (the user community). This is mandated by a complete reversal in the economics of design in computer technology. The exploding diversity of applications cannot be supported by manpower intensive development without eradicating the benefit of low-cost manufacturing. The burden of design diversity must be assumed by design automation and design evolution must become symbiotic to the evolution of application knowledge.

Metacybernetics is the marriage of two software engineering approaches. The first is the extension of computer interpretation to highest possible levels, making the production of application metacomputers, hence application languages, a straightforward process of application software engineering. The second is a technique of genetic reproduction deriving from compiler-compiler technology, replacing the surgical process of software modification with evolutionary resynthesis -- a process of reorganization which integrates a new whole, i.e. a mutation* rather than a modification. While both of these approaches have been in evidence on the fringes of computer science for about two decades, they could not progress beyond the R & D phase without enabling computer architecture.

1.0 A STRATEGY OF ADAPTIVE ENGINEERING

Metacybernetics involves an adaptive or experiment-oriented strategy of software-driven cybernetic development. The newly familiar phrase "rapid prototyping" is a strategy of engineering "problem-solving" programming which places the highest premium on quick solutions to engineering uncertainties. Engineering needs have always been the driver of higher-level languages and rapid-feedback approaches such as time-sharing, which reduce programming (hence experiment) time.

The inherent requirement of engineering problem solving is the ability to rapidly experiment with and evolve designs in the face of uncertain requirements. But traditional software practice cannot tolerate uncertain requirements, because they are the "initial conditions" of the historical development process.

Early-Specification Strategy - The early development of formal functional requirements defining software to be developed is a

---

* "mutation" is used to emphasize the context of evolution in which it is the generic term for offspring.

necessity _only_ to support the historical division of labor between diversified engineering and centralized programming. It has focused on the _end-item_ functions of software to be developed from scratch by the centralized programmers to fit specific requirements defined a priori by engineers. The programmer's responsibility has been to meet the stated requirements, whereas the engineer's responsibility has been the accuracy and completeness of requirement definition.

The achievement of effectiveness of software developed by such strategy requires clairvoyant engineering. Since programmers have been uncoupled from the responsibility for end-item effectiveness by the explicit statement of functional requirements, software designs do not focus on end-item flexibility as a prime directive. Rather, program efficiency has been usually assumed as the measure of programming quality. The net result has been high development cost, efficiency overkill and end-item inflexibility. These factors have caused the development part of the software life cycle to be very long and manpower intensive because they greatly extend the "engineering design feedback loop".

The major influence on life-cycle cost results from real uncertainty of requirements. Due to the experimental nature of engineering, software effectiveness is achieved only after much adaptive rebuilding. In effect, the development part of the life cycle never ends. What is often called maintenance is in fact, _surgical_ development -- not with the benefit of _resynthesis_ (the natural reorganizing characteristic of evolutionary mutation), but constrained by the limitations of the prototype design.

The early specification of functional requirements coupled with labor-intensive programming techniques (low-level languages) often leads to a misapplication of _functional description_ as _design structure_, resulting in overspecialized end-product logic. Implementation is specialized to the "superstructural" aspects of the problem delineated in the engineering requirements, even though the "substructure" methodology of the final product is inherently generic to broad classes of engineering problems. The early binding of formal specifications, usually coupled with high-pressure development schedules, motivates a _plunge into specialized detail_ followed by a _superstructure freeze_ followed by _surgical design modification_, etc. The end-product structure cannot be readily adapted because the development process does not preserve the orthogonal cleavage between specialized functional superstructure and generic method substructure.

The overriding influence on the character of the end-product is the sacrosanctity of the functional requirements. In engineering problem solving, where requirements are known to be uncertain, an alternate fundamental approach has evolved, coupled with engineering computing

decentralization -- the emergence of late-specification or "tool first" software strategy.

Late-Specification Strategy - A "tool first" approach shifts the focus away from the early specification of detailed functional requirements toward an adaptive knowledge base supported by a tool system to provide leverage in problem adaptability. While early-specification strategy, in order to fulfill programmer needs, emphasizes rigorous definition of functional requirements, late specification strategy utilizes criticalized models* of functional tasks as tool development test beds.

To maintain flexibility, the view is taken that functional requirements are but exemplary concepts of an uncertain spectrum which must be accommodated in its entirety. The emphasis therefore shifts from the diversified specifics of the functional requirements to their underlying unified methodologies as the only aspects of the stated requirements that can be known a priori with sufficient certainty. The resultant systems therefore have a built-in character of adaptability in order to deal in a timely manner with specific requirements as they become known.

The high leverage of late-specification tool strategy arises from the focus on the unity of engineering endeavor (e.g., mathematical tools) and the means of adaptation -- metaphoric programming languages -- unified media for distributing the burden of diversity to the diverse pool of user engineers. Although engineering is highly diverse and volatile, it is nevertheless unified and stable in its base of mathematical tools. These common tools can be applied very rapidly to specific engineering problems, without significant burden to the engineer, via "highest-order" languages.

2.0 ORIGINS AND CONCEPTS OF META DESIGN

Meta design employs metalanguages (language-describing languages) to automate the production of metacomputers from the highest order of design descriptions. The metalanguages are themselves processed by metacomputers produced in the same fashion from parent metalanguages processed by parent metacomputers. This concept is a form of "genetic recursion" that is characteristic of compiler-compiler evolution.

---

* contrived extreme requirements defined by worst-case and sensitivity analyses to illuminate the factors most sensitive to change and the modes of available leverage.

Metalanguages and Metaphoric Application Languages

Assumption of Generic Design Substructure - Compiler-compiler metalanguages (Refs. 2-6), deriving originally from BNF (Backus Naur Form, Ref. 7) are much higher-order languages than customary procedural languages. Since they derive from a language of pure description (BNF) which minimizes the amount of detail to describe a (language) design, they are examples of "highest-order" languages of the most universal kind. But, unlike BNF, they are used as metaphoric programming media to produce compilers from descriptions of the languages they are to compile. The language descriptions they provide to the compiler-compiler only specify the "tip of the iceberg" of the compiler that is produced -- that portion of its design that is specific to the subject language. The generic substructure of the produced compiler, which makes up the major portion of its design structure, is assumed by the compiler-compiler. It constitutes the implicit semantics of the metalanguage.

It is the assumption of this generic design structure into the language production medium, that produces the very-high translation leverage that is characteristic of compiler-compilers. This enables language design and implementation to be partitioned into a minor portion that is language-specific and a major portion that is generic to the family of languages encompassed by a specific metalanguage. The minor (language-specific superstructure) portion is the design, which would be developed by a user of the compiler-compiler. The unseen major (language-generic substructure) portion is the meta design -- a portion of the design of the compiler-compiler, produced as a mutation via evolution of a parent compiler-compiler. Thus meta design always precedes design, by at least one generation.

Diversification of Meta Design - The principle of meta design -- assumption of the generic portion of a design by the design medium -- can be used to achieve very-high programming leverage in any application. The result is the production of application language metacomputers, which assume in their substructure the generic tools of the application. Since most advanced programming currently involves expertise in two or more disparate programming languages (e.g., FORTRAN and assembly language), it is not unreasonable to expect meta design, employing metalanguages and subject languages, to be easily adopted by programmers. But it is doubtful whether general purpose meta design, via universal metalanguages will span the knowledge gap between metalanguage technology and application technology in a single division of labor. Because of current decentralizing economics it is rather more likely that meta design will become highly diversified from a central "trunk" of meta-meta design with branches of meta-application design having sub-branches of application design, etc.

This will involve the application diversification of the production of application language metacomputers -- a "trickle down" process presuming an evolution of special metalanguages, specific to special language families, mutated from more general metalanguages specific to families of language families, etc. This enables the specialized processes of end-product design, which must deal directly with the diversity of application logic, to be distributed to decentralized application environments while the "trunk" technology of universal meta design remains unified and centralized as a host technology.

Highest-Order Metaphoric Languages - A highest-order language is one that minimizes the amount of detail in the expression of a design. In the expression of a scalar algebraic formula, FORTRAN could be classed as highest order. But in the expression of a vector or matrix formula, FORTRAN would not qualify because extra information is necessary. A highest-order language may rely upon metaphoric technique to relate a design abstract in problem-description context (e.g., an engineering problem model) to a completely detailed design structure in program-execution context (e.g., a problem-specific algorithmic program) enabling the former to produce the latter via automation. Metaphoric technique fuses the two different contexts into one conceptual context of understanding, resulting in the simplest and most understandable description of a design. Designs are abstracted to their non-assumable essence (problem description) by submerging assumable detail (solution tool structure) into the production medium to be integrated into final product substructure via automation.

Metalanguages such as BNF, which are mainly used only to describe language designs, are good examples of highest-order languages because they are languages of pure description; there is no concern about solution substructure. But algorithmic languages such as FORTRAN, APL, or ADA do not generally qualify because algorithms, by nature, are solution methods, i.e., substructure. Designs are not conceptually framed in algorithms, but in problem metaphors which invoke algorithms as solution substructure. The key distinction between metaphoric languages and algorithmic languages is the degree to which the language emphasizes and enforces the separation between problem definition (metaphoric superstructure) and solution mechanics (algorithmic substructure).

Application Metaphoric Languages - The era of second generation computing was a time of great experimentation with metaphoric languages aimed at simplifying application programming. Representing the opposite extreme from machine-oriented assembly languages (metaphoric to hardware), these languages were metaphoric to a class of conceptual problem descriptions, with no direct reference to any hardware. They originated from the applications themselves, often as symbolic input to generalized problem programs, but subsequently evolved into problem describing languages containing "solver metaphors" to invoke built-in

problem solving tools. Notable examples were the continuous simulation languages DYNAMO, MIMIC, CSMP and CSSL; the discrete simulation languages SIMSCRIPT and SIMULA; and the synthetic calculus languages, SLANG and PROSE.

Such languages have demonstrated overwhelming advantages for software prototyping and the direct application of advanced mathematics. But they have been heavily penalized by historical architectures which do not support high-level interpretation efficiently. To submerge most of the complexity of programming, such languages employ dynamic memory-resident data bases as their operand environments, necessitating interpretive data management in execution. The historical role of the compiler as a pre-interpreter is greatly diminished, since compilers can only pre-interpret program formulations that are statically defined.

Interpretive Metacompilers - The metalanguages of the syntax-directed compiler-compilers were also developed and evolved during the 2nd generation and used to implement and evolve application-oriented languages. A key precedent to metacybernetics was the fact that that the metalanguages were often first implemented via interpretive techniques or were used to produce metacomputers (translator-interpreter combinations) rather than host-code compilers, because the development of interpreters is simpler and more amenable to a formal theoretic approach than the development of host-code compilers. The interpreter interface could be adapted to accommodate the subject language, easing the burden of translation.

But to become production tools on historical mainframes, compiler-compilers and the subject compilers they produced had to be wedded to the host-hardware by highly tailored code generators. This final phase of translation, precluded by interpretive systems, tended to dominate the compiler-development process, making the part that compiler-compilers could play, somewhat superficial. The upshot was that compiler-compiler technology, although a bright spot in software research, did not flourish in historical machine environments. It would only prove superior to the ad hoc approach if the hardware-software interface could be adapted.

Genetic Recursion - Another key precedent to this technology was the fact that the compiler-compilers were themselves metaprogrammed in their own metalanguages. Hence they could be reproduced in an automatic process analogous to ontogenesis -- the biological development of individuals of a species. They evolved, generation to generation, by parental reproduction (paralleling birth and growth) programmed by modification of parental metaprograms (paralleling genes); instead of being surgically modified and evolved in the manner of historical software development. Thus the development of

compiler-compiler technology paralleled natural genetic evolution, whereas historical software development did not.

The significance of this ontogenesis parallel to the future direction of cybernetics can hardly be overstated. Carried to its logical extensions through metacybernetic technique, it enables small, application-oriented teams to adaptively evolve giant software systems. It eliminates the extensive manpower cost and entropy gain of surgical maintenance and frees software dependent organizations from the shackles of software obsolescence.

Architecture Requirements

The primary barrier to the natural evolution of metaphoric languages is the lack of a means of implementing them via high-level, high-speed programmed interpreters. Historical architecture sacrifices the high speed potential of semiconductor logic in low-level-microcode interpreters emulating low-level virtual machines which serve as the historical software platform. Software interpreters built upon this platform are handicapped by two or more orders of magnitude in speed. Consequently, interpretive technologies like synthetic calculus (Refs. 8,9) are unable to be deployed to full advantage, and software design remains frozen at the level of programming achieved in the fifties, epitomized by FORTRAN.

The essential process of genetic evolution is only one step beyond the current state of the art of compilation technology. But <u>this step involves the ability to evolve the substructure under the compiler</u> -- the metacomputer instruction set and interpreter logic. With metacomputer host architecture as a foundation, it is feasible to design, generate and evolve metacomputers using BNF-type metalanguages.

3.0 METACOMPUTER DESIGN SYNTHESIS

The goal of metacomputer host architecture is the facilitation of metacomputer development by programmers and engineers who directly support applications. The aim is to rekindle 2nd generation drives toward application language evolution, extending beyond 2nd generation limits to the evolution of knowledge-based expert systems, thereby promoting symbiosis in the automation of man-machine problem solving. The key to this symbiosis is an understanding of how the application development process naturally unfolds, and provision of a support environment which facilitates and lubricates this process.

Metacomputer Form and Genesis

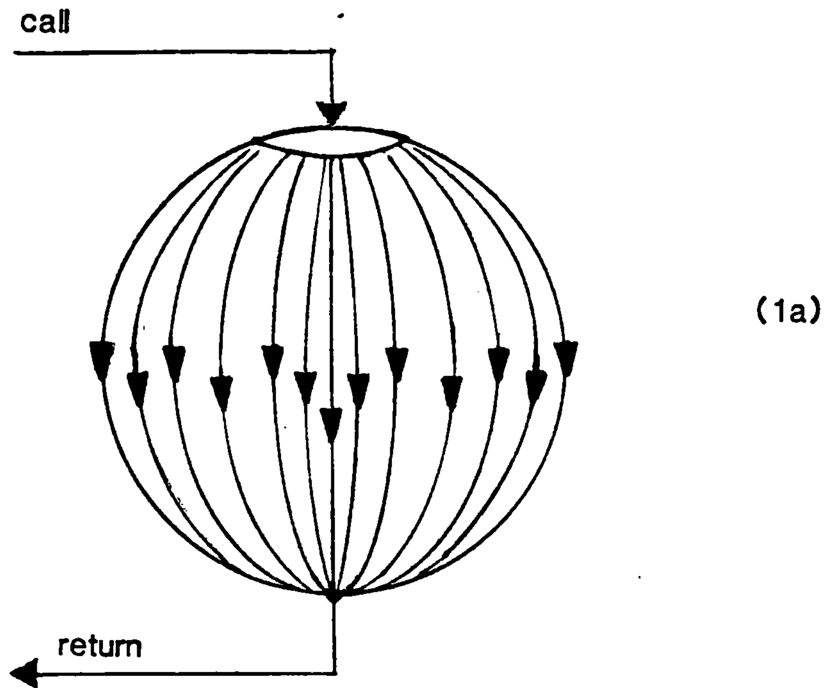
The Ubiquitous Function Bundle -- A metacomputer interpreter is logically simpler than a language compiler. Its characteristic structure is that of of a primal set (alphabet) of primitive functions (instruction logic in hardware). This conceptual structure has the same form in all modalities of automatic logic, and is particularly well suited for adaptive evolutionary change. The isomorphisms of Figure 1 characterize the simplicity of a level in a logic control hierarchy. One is that of a "Chinese lantern circuit" (Figure 1a). The other is a "carousel" (Figure 1b). The unit at the top of these structures represents a multiway switch, characterized by the FORTRAN computed GO TO, that decodes the incoming instruction stream (a set of integer switch tokens) and selects one of the vertical paths or "strands" along which the execution impulse moves, each representing a function. Once selected, the execution impulse travels down the strand through the bottom and returns to the higher control level of the hierarchy. Henceforth this isomorphism will be called the "function bundle".

The Genesis of Metaphoric Languages - Metaphoric languages usually evolve from the redevelopment of application software packages. They arise as symbolic grammars for describing executive programs that control application function bundles. The compilation of such languages is a simple assembly-type translation if the function bundles have a one-to-one relationship to the grammars. This is often the case because the function bundle is created before the language, as a set of tools pertaining to the application, and the language evolves to fill a user need for simplified problem expression.

Application function bundles usually evolve from the breakup of one or more monolithic application programs during later generation software projects, such as rehosting. The evolution from a fixed monolithic program structure to a set of configurable utilities arises out of the experience of attempting to apply old programs to new problems for which they were not specifically designed. In engineering applications this is a frequent requirement owing to the diversity of problem configurations (e.g., modeled engineering systems) within knowledge classes. This motivates a desire on the user's part to have a set of functional tools that can be rapidly synthesized into a tailored configuration for any arbitrary problem within the class. The more diverse are the applications in the class, the more the need for a metaphoric language is asserted. A natural division of labor develops between problem modeling (syntax) and tool development (semantics).

Metacomputer Evolution

The creative process of application metacomputer development advances heuristically from idea to trial to idea, encompassing many



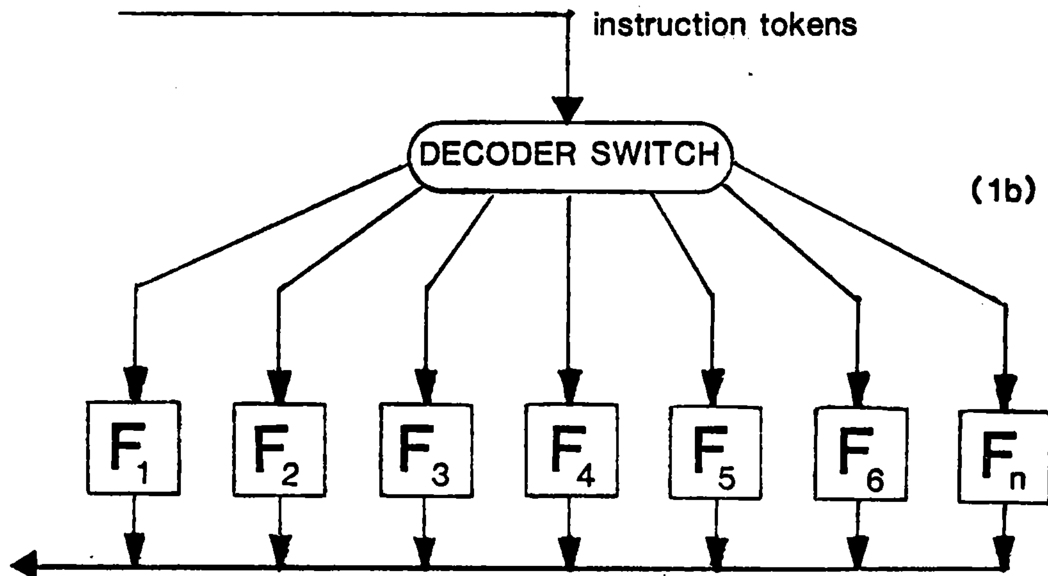


Figure 1. Function Bundle Isomorphisms

stages of evolving design in which each stage is a functioning, usable system. Each design is a resynthesis of its predecessor, involving a reintegration of structure, proceeding from particular to general. The modes of evolution involved in this process must be directly supported by the host environment.

Vertical Integration - Vertical separation of volatile and diverse problem definition from stable and unified mathematical tools produces a major advantage that evolves metaphoric languages. This is achieved by abstracting programming and data processing tasks to higher levels of definition, thereby reducing programming labor. Detail is reduced by integrating many low-level operation descriptions into fewer high-level operation structures. But in historical environments, this process is deterred by gross penalties in performance, because software evolution downward into the high-speed domain is prevented.

Hardware design has evolved vertically upward as chips changed from "letters" and "words" of electronic design to "sentences" and "paragraphs". But the domains of hardware and software evolution are not complementary, and do not integrate. Instead they are both frozen to the conceptual design of the original computers which has become the floor and ceiling isolating one design domain from the other. While hardware continues to advance through physical innovations of the eighties, application software is frozen in the logical technology of the fifties. A frozen software/hardware interface is the ultimate defeat to vertical integration because it blocks the mobility of evolution, preventing recurrent subassemblies from developing independent existence as stable tools which could be internally evolved in firmware or hardware to enhance performance.

A tool can have internal mobility in that it can evolve independently of its use so long as its statement of use, i.e., instruction interface, does not change. Its logical structure can be completely changed without altering its function. Thus evolution is permitted within a program hierarchy without disturbance of the hierarchy. This principle of interchangability is one of the properties of a metacomputer tool.

Complementary Tool Alphabets - But a tool must be more than interchangable. It must be useful as a complementary unit or primitive of description of the knowledge process of which it is a functioning part. Therefore it must have utility and universality. It must be a "primal" building block that appears in multiplicity within higher-order descriptions, either explicitly or implicitly. As a primal entity or element, it is stable in its role as a general subassembly, i.e., its evolution potential is minimal.

The complementary aspect is a property of the elements of a design (description) alphabet -- a set of independent parts designed to

operate as a matched set in combination to form higher wholes (i.e., whole designs). The quality of complementarity is measured indirectly by the synergistic leverage of combination -- the degree to which integrated whole designs are superior to non-integrated combinations. The hierarchic structure of such combinations enables the size of the description alphabet to be very limited. The property of complementarity distinguishes a metacomputer from an ordinary set of processing functions.

Metaprograms - A metaprogram is distinguished from a conventional procedural program in that it does not necessarily describe the semantic procedure that is executed by the hardware, but describes the essence of control information which completes the execution specification of the semantic procedure. It may, like a procedural language program, describe an analog or metaphor of a semantic procedure that is transformed into the procedure. Or it may indirectly define an input data case for a semantic procedure imbedded in the metacomputer. In this case the metaphorical expression of the problem is unrelated to the structure of the executed procedure.

Another distinction is that a metaprogram may presume a knowledge-base context, which completes the semantic specification. The same metaprogram syntax may therefore produce different semantics in different contexts. A knowledge base may have the form of a rule-oriented data base or it may be a mathematical topology as in the case of a synthetic calculus control frame (Ref. 9).

Precipitation of Application Design Alphabets - The primary problem of metacomputer design is the development of alphabets of tools for application software synthesis by metaprogramming. The genesis of tool precipitation from the breakup and resynthesis of application packages is only the first step in a process of trial and error providing the experience for alphabet generalization. Often, many of the elements of an evolved alphabet would not have been present in the original package, but would have precipitated out of the experience of use of the rudimentary alphabets of the breakup. The experience of synthesis of a diversity of applications is necessary to generalize the methodology of tool application, which indirectly defines and evolves the alphabet of tools.

Tools evolve from the generalization of methods. But the methods would not exist were it not for the tasks that they generically perform. The order of evolution must therefore be (1) tasks precipitate methods, and (2) methods are generalized into tools. A clear implication of this is that metacomputers must originate and develop in application-oriented environments.

4.0 METACYBERNETIC ENGINEERING

The aim of metacybernetics is the integration of design and evolution in cybernetic engineering. This requires a capability for small vertical teams of application engineers, software engineers, digital logic designers, and electromechanical designers to operate creatively together in a laboratory environment. The means of developing mature designs must be one of rapid prototyping, enabling promising ideas and inventions to be brought to maturity by accelerated synthesis and testing.

Design Evolution Mobility

The essential process of product design involves the choice of design parameter values to optimize some design attribute, such as performance or cost-performance, subject to constraints of application. Each design parameter represents a dimension of variation, an axis of proportionality which affects the design attribute. A design configuration is therefore tuned by choosing a specific combination of parameter settings. In order to provide a context for design evolution, there must be freedom for parametric variation. A context that facilitates such variation is said to have design-evolution mobility. This is the important prerequisite of metacybernetic design that is facilitated by metacomputer host architecture.

The Dimension of Primal Design - The significant advantage of the function-bundle structure is that the functions are modular, anonymous, and uncoupled as far as the overall structure is concerned (i.e., they are orthogonal to each other and to the overall structure), and the overall structure is an open-ended exchange facility. It may grow by the simple addition of more primal functions (strands on the lantern, boxes on the carousel), without disturbing the functioning of the whole. Thus, such systems have a dimension of evolutionary variation corresponding to primal-function design.

The Dimension of Function Physiology - Functions can be implemented in three different structures -- hardware, firmware, and software, which differ markedly in execution speed. Since there are many different levels of speed in each of these broad categories, each having different costs, we may think of this as another dimension of variation for design evolution -- the dimension of function physiology. We can choose to implement any function in any physiology. We may imagine an analogous picture of a Chinese lantern with thin fibrous wicks to pass fluidic impulses, varying strands of copper to pass electrical impulses, and fiberoptic strands to pass light impulses -- all combined in a variegated function bundle.

Physiological Evolution - One of the subtle benefits of hierarchic structure is its facility for evolution. H.A. Simon (Ref. 11) has illustrated with his famous watchmaker parable that hierarchies evolve faster than any other type of structure. The most important characteristic facilitating evolution is the "plug-in" concept -- standardized interfaces. This is most obvious in electronic chip design. Once the interfaces of a set of plug-modules have been designed, a design hierarchy comes into being. Thereafter plug modules having the same plug interfaces and performing the same functions may be substituted at will. Thus component modules may evolve without disturbing the hierarchy. This happened every time two physiologically different computers were designed for the same machine language, e.g., the IBM 709 (vacuum-tube physiology) and the IBM 7090 (transister physiology), and then interchanged. The same software would execute on both, albeit much faster on the 7090. The logical hierarchy was undisturbed. The stable logical hierarchy is therefore the fixed conduit for physiological evolution, and physiological evolution is the performance control variable for stable software.

The Dimension of Mutation - The appearance of a new species is a result of a reorganization of the hierarchy itself -- the system superstructure. This dimension of design variation is provided by the tools of genetic reproduction -- metacomputer generators. A metacomputer generator is a derivative of an advanced compiler-compiler which is itself an evolving metacomputer. It generates a complete metacomputer (assembler and interpreter) in a single computer run from a genic metaprogram described in a BNF-type metalanguage. The genic metaprogram is the mechanism of hierarchy evolution, and each run is a total resynthesis, equivalent to the ontogenic production of an individual of a species.* A change of the metaprogram results in a mutation, a member of a new species.

Performance Engineering

Having three dimensions of design variation to work with, the problem of performance engineering is to optimize the composition of the function bundle to match performance and cost requirements. For a given metacomputer primal structure, this amounts to optimizing the function-bundle metabolism -- the mix of function physiologies -- to optimize performance or cost-weighted performance.

The Exogenous Variables - The two primary variables are not free to be varied by the designer. They are the logic capacity of the primal functions (L), measured in operations or steps per function execution

*technically a clone since every compilation of the same genic metaprogram produces an identical copy.

(ops/call), and the application frequency (F) measured in calls per second. While the former is known from the design specification of a primal function, the latter is job-mix dependent and can only be approximated from empirical evidence. On the other hand, the relative frequencies of groups of functions can be estimated on the basis of theoretical considerations, and this is sufficient to determine initial physiological choices. Later, these choices may be "tuned" as the result of performance measurements.

Performance Optimization - The performance of a metacomputer can be optimized by strategic selection of primal function design and physiology so that the largest of the most frequently used functions have the fastest physiology while the smallest of the most infrequently used functions have the slowest physiologies. This rule can be expressed by the formula P=LF. Given estimates of the logic capacity and the application frequency (derived from experimental usage), this formula, computed for each primal function establishes a priority set for assignment of physiological levels to primal functions. This process defines the architectural structure of the function bundle, and establishes the priority of metacybernetic research as well.

Metacomputer Host Environment

The metacomputer host is a special architecture configured for the implementation and evolution of application metacomputers. It is therefore a metacybernetic engine, providing the same freedom of design in a total cybernetic sense that a compiler-compiler provides to a compiler designer.

Software Bifurcation - Rapid evolution is facilitated by two levels of software having widely different execution speeds (two to three orders of magnitude). The high-speed internal level constitutes the metacomputer kernel. The low-speed external level constitutes the application software. The latter can be regarded as data to the former. In its "frozen" form, implemented in ROM, the internal software becomes firmware.

The internal software is intraprocessor software, since it is directly executed by the hardware units, whereas the external software is interprocessor or network software. To the former, the image computer is a single hardware processor. To the latter, the image computer is the metacomputer -- there is no distinction of hardware units. In other words, the metacomputer may be partitioned among several processors configured in any suitable network.

In between the two levels is a mutual interface -- the metacomputer instruction set architecture (ISA), an architecture of function bundle switch tokens which serves as the interface to its metaprogramming

language. This is the executable image of the external software. It is generated for execution by an application syntax assembler.

The internal software, while its purpose parallels the historical role of microprogramming, should not be handicapped by the very-low level of programming, characterized by conventional microcode. Rather, it should implement the highest level of programming possible via direct hardware implementation. This permits the highest level of execution speed and the greatest flexibility for evolution.

Metacomputer Structure - A metacomputer has function-bundle structure. It consists of:

(a) an executive superstructure constituting the grammatical tree-logic of its ISA

(b) a function-bundle alphabet of token processing functions constituting the substructure of interpretive execution.

The token function alphabet is a generic set of interpreter tools which may be the same for all programming languages within broad classes or genera. The language species is therefore identified with the tree-logic of the grammar executive, whose structure is isomorphic to the ISA. Thus the ISA is the metaprogram of the grammar executive of the metacomputer. The switch token components of the ISA are isomorphic to the token function alphabet, hence the ISA token alphabet is the metaprogram of a genus of metacomputers, i.e., a class having the same substructure for all species of the genus but with varying grammatical superstructure.*

Metagenic Synthesis and Evolution - The process of metacomputer evolution, involving, as it does, three dimensions of mobility, also proceeds in essentially three levels of change frequency commensurate with the rates of identification of adaptation requirements. These levels of change frequency are cycles within cycles, where "first-order" changes are responsive to the most volatile needs for adaptation, "second-order" changes are responsive to needs brought about by first order changes, and "third-order" changes are responsive to needs brought about by first and second-order changes, etc.

Ahead of first-order changes are zero-order changes that are essentially outside of the design domain, but directly affect

---

* The hierarchal role of species superstructure to genus substructure parallels natural language, where a genus would be common sounds associated with an alphabet and a species of the genus would be the words and sentence structure of a specific language which uses the alphabet and its associated sounds.

first-order changes. These correspond to the differential variations in application metaprograms which utilize a given metacomputer. First-order changes, therefore, correspond to differential changes in metaprogramming language -- variations in the dimension of mutation. These can range from minor variations distinguishing dialects to major variations distinguishing genera. The gradient of first-order change is typically very large owing to the diversity of applications that the metacomputer host may address.

The mechanism of first-order change is <u>metagenic</u> synthesis, illustrated in Figure 2, in which the specialized essence of a design, defined in a genic metaprogram, complements the generic tools subsumed in the design medium to <u>complete</u> the design specification and generate the design superstructure, i.e., the executives of the metacomputer. The remaining categories of change (2nd and 3rd order) are limited to design substructure.

The primary agent of first-order change is the metacomputer generator. It compiles a language species metacomputer into the host environment by translating its genic metaprogram. The genic metaprogram contains highest-order specifications of the three most volatile elements of the metacomputer, its language syntax, its ISA, and its interpreter semantics, coded in a BNF-derivative metalanguage. The output of the generator composes the executable logic of the executives of the syntax-to-ISA assembler and the ISA interpreter.

Second-order changes are essentially changes in tool software/firmware of the metacomputer assembler and interpreter. These changes constitute variations in the primal-function dimension. The evolution of mathematical algorithms, such as the "solvers" of synthetic calculus, fits into this catagory. Since these tools are of the nature of utilities, they are typically universal to a language genus, and are consequently relatively stable in functional role. Another category of second-order changes is responsive to variations in the metalanguage of the metacomputer generator. Technically this is a first-order change since the metacomputer generator is itself a metacomputer, and would be used to metagenically reproduce a mutation of itself. Experience has shown that such changes are about as frequent as changes in the tools.

Third-order changes constitute vertical physiology evolution, dictated by performance requirements. Requirements for such changes are computed by the differential physiology formula P=LF, to compensate for second-order changes in the interpreter tools (L) and first-order changes in the statistical moments of the application mix of the metacomputer host (F). Changes of this nature would normally affect common substructures of many tools. Examples include matrix-inversion, synthetic differentiation, basic methods of numerical integration, etc., since these are utility processes used in families of software and firmware algorithms.

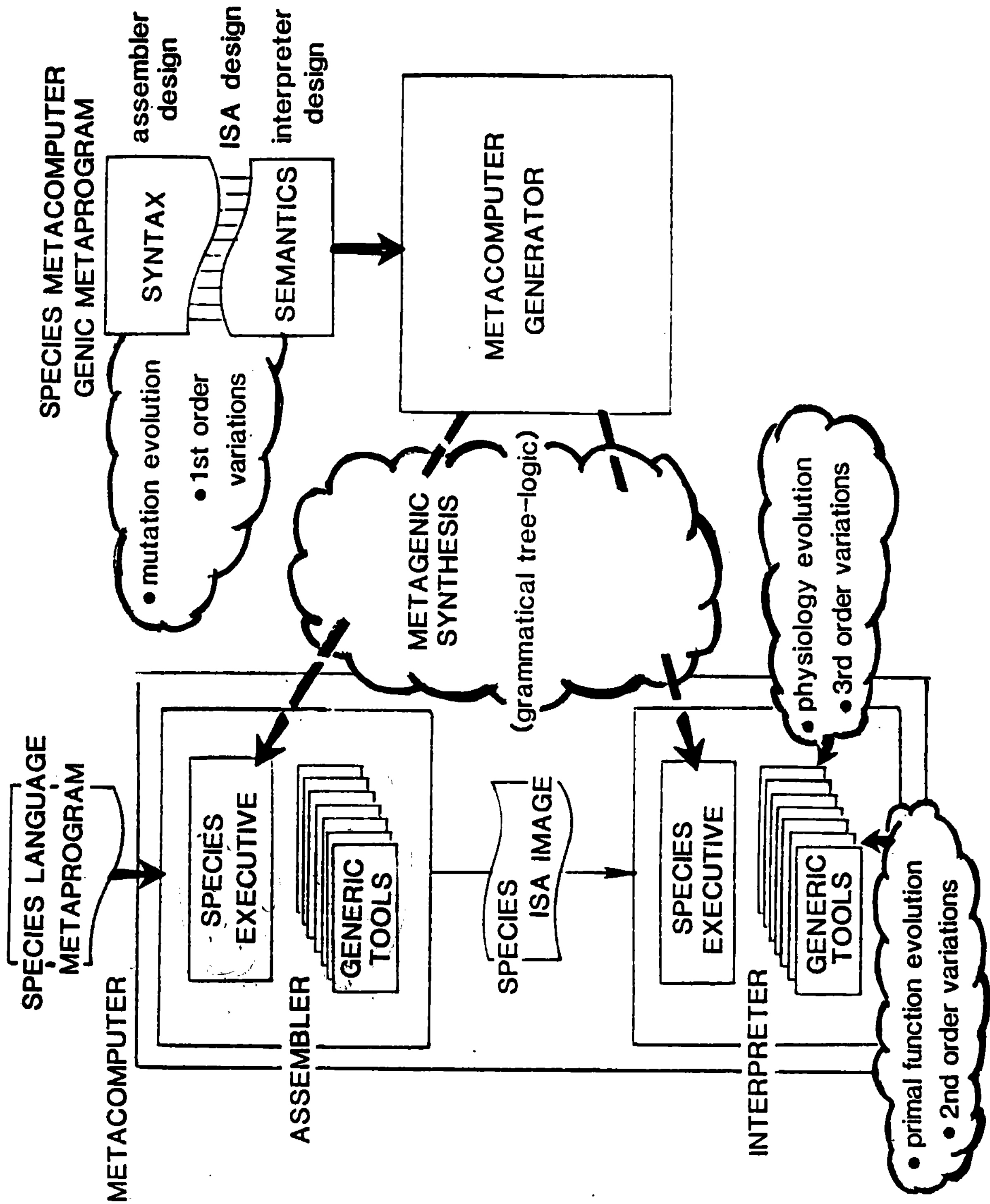


Figure 2. Metacomputer Structure and Evolution

5.0 CONCLUSION

Second generation software R & D originated highest-order language-directed techniques which demonstrated high leverage in software prototyping and evolution. But conventional low-level architecture eradicated the benefits of this leverage by not meeting it half way, i.e., by not enabling the firmware interface to vertically evolve into direct interpretation of high-order ISA's. Software technology has been blamed for its high cost, but the fault lies with frozen-software-interface architecture. Metacybernetics is the result of software R & D having its proper say in hardware architecture design.

Metacybernetics recognizes the non-industrial (i.e., non-centralizable) character of scientific research and development. It provides the means for diversification in design while at the same time providing the means for orthogonal unification of meta design. It provides a balance of formal and empirical approaches, using each to refine the other. Its utilization, however, will emphasize a cultural shift in the computer professions, as cybernetic design becomes as diversified as engineering.

The Burden of Diversity - There can hardly be any doubt that centralization in computing is a thing of the past. Vanishing hardware costs, a programming bottleneck, and exploding diversity of applications are mandating an upward and outward shift of cybernetic design, away from the centroid of computer science toward application programming domains. The gradient of diversity in applications is far too great to be centrally managed except strategically via the unity within the diversity. To develop viable designs, we must seek to find that unity, to limit our designs to it, and provide the tools for the extension of these designs by the application engineers. This clearly means that we must move from design to meta design. Metacybernetics shifts the prime role of design from a highly centralized division of labor, which has characterized the computer industry from its inception, to a more decentralized one coincident with current economic trends.

Metacybernetic Diversification - The acquisition by the computer of the knowledge bases of its user paradigms has been identified elsewhere (Ref. 12) as the crucial first step toward machine intelligence. But of more immediate importance than this lofty goal is the achievement, by the same means, of effective man-machine symbiosis in everyday science and engineering. More important than machine intelligence is the reduction of the entropy of diversity in information processing by the acquisition and evolution of the nexus of refined experience that constitutes engineering knowledge.

The metacomputer host is the unified kernel for late-specification metacybernetic diversification. Each distributed host may adapt to the discipline and personality of the scientific paradigm that it supports, becoming by evolutionary growth the permanent repository of the paradigm knowledge base. As a mechanism of automating corporate memory, it provides a means for "banking" the investment in knowhow of technological endeavors -- independent of the continuity of the personnel involved. Only by such a process of extracting and preserving the order of knowledge out of the chaos of experience, can we hope to defeat the entropy law.

REFERENCES

1. Speckhard, A.E., and Fleming, R.C., "The Aerospace Research Computer, A Reconfigurable High-Level Language Machine", Proceedings of the Workshop on High-Level Language Computer Architecture, Los Angeles, CA, October 7,8 and 9, 1981.

2. Carr, C.S., Luther, D.A., and Erdmann, S., "The Tree-Meta Compiler-Compiler System: A Meta Compiler System for the Univac 1108 and the GE 645" , ARPA RADC Research Report, University of Utah, March 1969.

3. Feldman, J. and Gries, D., "Translator Writing Systems," Communications of the ACM, February 1968.

4. Oppenheim, D.K. and Haggerty, D.P., "META5: A Tool to Manipulate Strings of Data," Proceedings of the 21st National Conference of the ACM, 1966.

5. Schneider, F.W. and Johnson, G.D., "META-3, A Syntax-Directed Compiler-Writing Compiler to Generate Effecient Code," Proceedings of the 19th National Conference of the ACM, 1964.

6. Schorre, D.V., "META II, A Syntax-Directed Compiler Writing Language," Proceedings of the 19th National Conference of the ACM, 1964.

7. Naur, P. et. al., "Report on the Algorithmic Language ALGOL 60," Communications of the ACM, Vol. 3, No. 5, pp. 299-384, May, 1960.

8. Thames, J. "The Evolution of Synthetic Calculus - A Mathematical Technology for Advanced Architecture", Proceedings, International Workshop on High Level Language Computer Architecture, Nov. 30, 1982, Fort Lauderdale, Fla.

9. Thames, J. "The Structure of Synthetic Calculus - A Programming Paradigm of Mathematical Design", (elsewhere in current proceedings).

10. Thames, J.M., "SLANG, A Problem-Solving Language for Continuous-Model Simulation and Optimization", Proceedings, ACM 24th National Conf. (December 1969) pp. 23-41

11. Pattee, H.H., Hierarchy Theory - The Challenge of Complex Systems, Geo. Braziller Pub., New York, 1973.

12. Feigenbaum, E.A., and McCorduck, P., The Fifth Generation, Addison-Wesley Publ. March, 1983.