The Price of Programmability

Michael Conrad

1. *Introduction*

Programmability and computational efficiency are fundamental attributes of computing systems. A third attribute is evolutionary adaptability, the ability of a system to self-organize through a variation and selection process. The author has previously proposed that these three attributes of computing are linked by a trade-off principle, which may be roughly stated thus: *a computing system cannot at the same time have high programmability, high computational efficiency, and high evolutionary adaptability* (e.g., Conrad *1972, 1974, 1985*). The purpose of the present paper is to outline the reasons for the trade-off principle in a manner which, though not entirely formal, is sufficiently detailed to allow for a well-defined formulation. We also consider the implications of the principle, first for alternative computer architectures suited to solving problems by methods of evolutionary search and second, for limits on the capacity of programmable machines to simulate nature and duplicate intelligence.

2. *Programmability*

Programmability is difficult to define outside of the context of a specific formalism, such as Turing machines, recursive functions, or high level computer languages. We can proceed by defining the concept of a program in terms of a specific formalism, and rely on the Turing-Church thesis to argue that this definition has an equivalent in every other formalism. Alternatively, we can make metaobservations about programs in all these formalisms. In particular we note that programs are rules, or maps, that feature finiteness of the number of symbols employed, discrete differences among

the symbols, and, in the action of the rule, discreteness in time. The classical definition of a finite automaton has all three properties, and we can regard the next state function and output function as comprising the program of the finite automaton. Programmability is the ability to prescriptively communicate a program to an actual system.

Definition 2.1 (Turing machine program): *We recall that a Turing machine is a finite automaton along with a potentially infinite memory tape which it can move and mark. A finite automaton may be defined as a quintuple* $< X, Y, Z, \lambda, \delta >$, *where* X *is a finite set of inputs (tape symbols),* Y *is a finite set of states,* Z *is a finite set of outputs (tape symbols and moves),* λ *is the next state function* $(X \times Y \rightarrow Y)$, δ *is the output function* $(X \times Y \rightarrow Z)$, *and we assume a discrete time scale. The next state and output functions (the transition functions) comprise the program of the Turing machine (or of the finite automaton).*

Definition 2.2 (simulation): *System* S_1 *will be said to simulate system* S_2 *if it executes the same transition function from initial state to final state and output. We do not require that all aspects of* S_2*'s states be represented in* S_1*'s states and we make no assumption about the fidelity of intermediate states.* S_1 *and* S_2 *are weakly equivalent if they simulate each other (in this case all aspects of the state description must be included). They would be strongly equivalent if the intermediate states were faithful, that is, if the simulation would hold at every time step. We will say that a class of systems* $[S_1]$ *is equivalent to a class of systems* $[S_2]$ *if all the systems in* $[S_2]$ *can be simulated by some system in* $[S_1]$ *and conversely. The simulation will be called exact if the final state of* S_2 *can be precisely ascertained from* S_1 *and will be called approximate if it can be ascertained to an adequate approximation (any reasonable definition of "adequate" will do).*

Rules (or maps) that generate or describe the behavior of a system may be divided into finite and infinite types. A rule is of the finite type if it can be decomposed into a number of types of distinct operations (to be called primitive operations) that are applied to a finite set of distinct symbols (called primitive symbols). According to the Turing-Church thesis all such finite-type rules are equivalent to Turing machine programs in the strict sense that each of the primitive rules from which they are built can be simulated exactly (that is, without approximation) by Turing machines. The converse is also true if the simulating system is universal. Interpreted in its strongest form, the Turing-Church thesis also asserts that, as far as presently known, all physically realizable maps can in effect be computed by Turing machines, that is, the systems described by these maps can be simulated

to an arbitrarily high degree of precision (assuming constraints on space and time bounds are ignored). Thus it should be possible to associate any infinite-type map that is physically realizable, such as the continuous maps of physical mechanics, with rules of the finite type. This distinction motivates the following definition.

Definition 2.3 (program): *A program in the strict sense is a rule that (when embodied) generates the behavior of a system, subject to the condition that the rule is of the finite type. This generalization of Definition 2.1 is justified by the assumption that each distinct element of any finite-type rule can be exactly defined in terms of the Turing machine programs. The maps describing the behavior of systems not of the finite type will not be called programs, but they may be implicitly associated with programs of the finite type (again by virtue of Turing-Church). These associated programs will be called implicit programs.*

We note that the concept of computation can be generalized from the well-defined concept of Turing machine computation in a similar manner. To the extent that all physically realizable dynamics can be simulated by Turing machines it is possible to assign an equivalent amount of Turing machine computation to their behavior. For example, it is possible to use digital simulation to express the amount of *equivalent* digital computation performed by, say, a turbulent fluid. The precise amount of equivalent computational work will depend on the model of computation used for the simulating system and on the efficiency with which the time and processor resources of the simulating system are used. The objection may be raised that computation should be defined in terms of purposes, such as the solution of problems. However, it is clear that Turing machine behavior may serve no useful purpose, aside from generating the particular behavior in question. Moreover, arbitrary continuous dynamical processes could conceivably be incorporated into computing machines as primitive operations in a manner that enhances the utility of these systems for solving problems. Turing machines and models of digital machines generally are particular models of computation. They are particularly useful as reference points for evaluating the amount of computational work performed by arbitrary dynamical systems, not as delimiting the class of behaviors admitted to be forms of computing.

Having defined the concept of program we are in a position to define programmability, and in particular the important concept of structural programmability.

Definition 2.4 (effective programmability): *A real system is (effectively) programmable if it is possible to communicate desired programs to it in an exact*

manner (without approximation) using a finite set of primitive operations and symbols.

The finiteness of the set of primitive operations and symbols is tantamount to the requirement that the user's manual be finite. This is what allows the programmer to communicate the rules he conceives exactly, that is, to exert complete control over the rules governing the machine. It is undoubtedly possible to omit the subjective state of the programmer, what he desires to impart to the computer, from the definition. However, our intuitive concept of programmability is strongly connected to our ability to express algorithms directly as programs, using computer *languages*. I have used the term "effective" to emphasize the completeness of the control exerted by the programmer. This sense of complete control is of course also subjective. In fact, it is impossible in general to prove that programs express desired algorithms, and as programs become large it is inevitable that they will be in some measure incorrect (see Avizienis *1983*). A concept of approximate programmability might fit real-world phenomena better than effective, or exact, programmability. However, the concept of exact programmability fits better to the objectives of the majority of programmers.

Programmability can be achieved in three ways. The first is to construct an interpreter, or universal algorithm, that can read and follow any particular program. It is necessary to construct a real system capable of executing this algorithm. The second way is to build a physical realization of the formal system—that is, a machine whose elementary operations and states match the primitive operations and symbols of a programming language. This provides a mechanism for communicating a universal rule to a physical machine. The third way is to use a compiler, or algorithm, which converts the primitive operations and symbols of a psychologically convenient language into primitive operations and symbols that match the operations and components of the machine. The latter two methods are based on the fact that the program, either directly or indirectly, can be explicitly represented in the structure of the physical system, that is, in the state settings and connectivity of its components.

Definition 2.5 (structural programmability): *A physical system is structurally programmable if it is effectively programmable and if its program is mapped by its structure.*

Effective programmability does not necessarily entail structural programmability. In fact it is possible to show that a physical system can be structurally nonprogrammable and at the same time effectively programmable (Conrad *1974a*). To do so it is necessary to construct a structurally nonprogrammable system that is computation universal. If it is computation universal it must be able to support an interpreter. However, all present-day artificial computers are structurally programmable.

Structural programmability may be viewed in terms of the two branches of automata theory—behavior and structure. Behavior theory deals with the capabilities of machines from either a state or language point of view, while structure theory is concerned with either synthesizing systems from primitive components or analyzing them into primitive components. These two branches are logically independent. A combination lock is a finite automaton, but it is not ordinarily decomposable into a base set of elementary-type components that can be reconfigured to simulate an arbitrary physical system. As a consequence it is not structurally programmable, and in this case it is effectively programmable only in the limited sense that its state can be set for achieving a limited class of behaviors. A digital computer used to simulate a combination lock is structurally programmable since the behavior is achieved by synthesizing it from a canonical set of primitive switching components. We shall see that it is usually necessary to pay for this decomposability and universality in terms of the potential efficiency with which the material resources of the system are utilized.

3. *Programmability and Efficiency*

Computational complexity usually refers to the number of time and processor resources required to solve a problem (Kuck *1978*). I will use the term computational efficiency to refer to the effectiveness with which resources are recruited for solving problems. To compare the ultimate problem-solving power of widely different types of systems, including systems that are structurally nonprogrammable, it is convenient to take the number of processors as the number of particles in the system. We could also consider the costs of composing these processors into an integrated system; however, this cost has negligible impact on our analysis and is in any case highly contingent upon the broader support systems available. Furthermore, we will for simplicity focus on number of particles rather than time, considering the comparative potential computational power of systems over equal intervals of time.

Particles contribute to computation only insofar as they interact with one another. If more interactions are recruited for computing, fewer particles will be necessary. As a consequence we define efficiency in terms of interactions rather than particles, using the laws of physics to translate, when necessary, from number of interactions to number of particles.

Definition 3.1 (computational efficiency and computational complexity): *The computational efficiency of a computing system (denoted by ε) is the*

number of interactions used for computation relative to the maximum number possible in a system consisting of the same number of particles. The computational power of a computing system is measured by $P = n\varepsilon$, where n is the number of particles it contains. The computational complexity of a problem is the minimum number of resources required to solve it (for the present purposes expressed in terms of number of particles, time, or both).

According to current force laws, n^2 interactions occur in a system of n particles. A digital computer simulating such a system must calculate all these interactions. In practice the number of interactions among the particles in the digital computer is very much less than n. If the computer is effectively programmable it must operate in a sequential mode, otherwise unanticipated conflicts would always be possible. Roughly speaking a computer consisting of n processors can support at most an n-fold speed-up (e.g., Arbib *1969*). The underlying assumption here is that the definition of a processor in the user's manual should be scale invariant—it should not change as more processors (therefore more particles) are added to the machine. The amount of information processing carried out by a physical system freed from the constraints necessary to support programmability is thus potentially much greater than the potential information processing performed by a system not so constrained.

It may be objected that the nonprogrammable system is not demonstrably processing information. However, we can recall (Section 2) that an equivalence can be established between computing and arbitrary physical dynamics. A digital computer that simulates a nonprogrammable system, such as a turbulent fluid or a DNA molecule, is indubitably processing information. The difference between the digital computer and the simulated system is that the former has a universal simulation capability, whereas the latter may have a limited simulation capability. Restricting the potential number of interactions that can contribute to computing can be justified only if the concept of computing is tied to a particular class of machines. If the equivalence between computing and dynamics is admitted, then n^2 must be the upper limit of interactions that can contribute to computing (Conrad *1984*).

Many constraints are pertinent to the actual computing power achieved by both structurally programmable and structurally nonprogrammable systems. Some of these constraints are connected with restrictions on static and dynamic degrees of freedom of the system (Pattee *1973*). Some are connected with constants of nature, such as the speed of light, Planck's constant, and Boltzmann's constant. When a computing system becomes sufficiently large, the speed of light should be taken into account. Efficiency, ε, would then assume a dependence on size. However, the following model provides a

useful first approximation to the comparative computing potential of structurally programmable and structurally nonprogrammable systems (see also Conrad and Hastings *1985*).

Consider a structurally programmable system in which the processors (including both switches and wires) behave according to the specifications in a finite user's manual. As before the system consists of n particles ($1 \leq k \leq n$). It is straightforward to specialize to the probably unrealistic limit in which each manual-defined processor is a single particle. For simplicity we will assume particles are added k at a time (that is, in integer units of processors). The number of interactions that can contribute to computing then increases with the number of particles by at most $C(n/k)k^2$, where n/k is the number of processors and C is a constant representing the number of potential contacts that a processor can have and nevertheless operate according to its definition in the user's manual. As k increases the potential contribution of the processor to computing increases. Usually only a small fraction of the k^2 interactions in a processor will be utilized, however. Furthermore, the factor Cn/k is an upper estimate since effective programmability is in general lost if all processors are used at once even if the machine is effectively programmable when operated in sequential mode. The efficiency of structurally programmable systems thus scales at most as $\varepsilon = Cnk/n^2 = Ck/n$, and, in the limit of each particle being a manual-defined processor, as $\varepsilon = C/n$. This means that the efficiency of structurally programmable systems decreases as the number of particles in the system increases.

By contrast, structurally nonprogrammable systems can achieve an efficiency of $\varepsilon = 1$. This is the case when none of the interactions are suppressed. However, in this case the system has no flexibility other than that associated with the choice of initial conditions. If constraints are imposed on it to tailor its behavior for a particular task this must reduce the number of interactions. Maximum flexibility is achieved when the allowable number of variations is greatest. According to the binomial theorem this occurs when the number of interactions is $n^2/2$ (Harary and Palmer *1973*). The efficiency of structurally nonprogrammable systems constrained for maximum evolutionary flexibility thus scales as 1/2, independent of the size of the system.

The situation can be summed up in the following definition and theorem:

Definition 3.2 (evolutionary flexibility): *The evolutionary flexibility of a system is the number of possible variations on the pattern of interactions among its constituent particles compatible with the constraints that define the class of systems to which it belongs.*

Theorem 3.1 (efficiency versus programmability and evolutionary flexibility): *The efficiency of a system constrained for structural programmability scales at most as Ck/n, where n is the number of particles, k is the number of particles per manual-defined processor, and C is a constant. The efficiency is less if the system is further constrained to run in a sequential mode, in general a requirement for effective programmability. The efficiency of a structurally nonprogrammable system organized for maximum evolutionary flexibility scales as $1/2$, independent of the number of particles it contains. Thus the efficiency of structurally nonprogrammable systems is potentially $n/2Ck$ larger than that of structurally programmable systems when the former are tuned for maximum flexibility and the latter are run in the parallel, least programmable mode. The advantage of structural nonprogrammability increases still further when the programmable systems are run in the sequential mode, and when any of the k^2 interactions with a processor are suppressed. The potential computational power of a structurally programmable system consisting of n particles is a constant, Ck, as compared to $n/2$ for a maximally flexible, structurally nonprogrammable system.*

To complete the picture, we compare these computational capabilities to the computational resources required for solving problems. We recall that problems are commonly divided into two broad categories, those with polynomial and those with exponential growth rates (Garey and Johnson *1979*). We take polynomial-type problems to be those in which the number of computer resources, here the number of particles, required to solve the problem grow as a small polynomial function of problem size—say as n^2, where n is a measure of problem size. The resources required to solve an exponential-type problem increases combinatorially, say as 2^n.

If the size of a structurally programmable system is increased by 10^{10} and if all of its resources could be recruited in parallel with perfect efficiency, it could potentially increase the size of n^2 type problems it could handle by a factor of 10^5. This is if we consider the computing resource to be the number of manual-defined processors, as is usually done. If we consider the resource to be the number of interactions, we could conceivably take advantage of some of the contacts among processors and some of the interactions within them. Even so, the number of interactions, Ckn, scales as the number of processors, and in the limiting case in which all particles are processors $C = k = 1$. In structurally nonprogrammable systems, however, the number of interactions increases much faster than the number of processors, in fact as the square of this number when the processors are identified with particles. Thus the increase in the computing potential of a structurally nonprogrammable system is in this case 10^5 times better than

the increase in computing potential of a structurally programmable system run in a parallel mode. Summing up we have

Theorem 3.2 (structural programmability versus computational complexity): *The size of an n^2-type problem that can be solved by a structurally programmable system consisting of n particles increases by a factor that scales at most as $n^{1/2}$, even if all interactions in the system can be brought to bear. By contrast, the size of the problem solvable by a structurally nonprogrammable system with maximum evolutionary flexibility increases by at most $n/\sqrt{2}$.*

Theorem 3.2 implies that structurally programmable systems cannot keep up with polynomial growth rates in problem size, whereas structurally nonprogrammable systems can in principle keep pace with these growth rates. So far as is known, no system can keep up with exponential growth rates. If a physical system could keep up with exponential growth rates we would have to give up the idea that it is simulatable by digital computers in polynomial time for even the smallest time slices. If it were so simulatable we could solve these exponential time problems in polynomial time by a constant factor speed-up of the digital computer, contradicting the assumption that the problem is of the exponential type. It is of course possible that such nonsimulatable systems exist. This would not contradict the Turing-Church thesis, however. In its strongest form Turing-Church requires all physical processes to be simulatable; but it places no polynomial growth limitation on the number of computational resources required for the simulation.

4. *Evolvability and Gradualism*

We now investigate necessary and sufficient conditions for evolutionary adaptability. Evolutionary flexibility and gradual transformability (to be defined below) play an important role. In the next section we show that the problem of ascertaining whether a structurally programmable system is gradually transformable is in general unsolvable and that the class of structural changes for which the problem is solvable is much greater for structurally nonprogrammable systems. As a consequence programmable systems are not as effectively structured for evolution as nonprogrammable systems.

Definition 4.1 (evolutionary adaptability): *A system is evolutionarily adaptable (or evolvable) to the extent that it can utilize variation and selection mechanisms to survive in uncertain or unknown environments. Survival means continuation of function at an acceptable level. Evolutionary adaptability may be operationally measured by the uncertainty of the most uncer-*

tain environment that a system can tolerate on the basis of variation and selection mechanisms alone. (Entropy measures on the environment transitions, treated as Markov chains, are suitable for this purpose. See Khinchin 1957; Conrad *1983*).

Definition 4.2 (gradual transformability): *A system is gradually transformable if it undergoes small changes in behavior in response to at least one elementary (or noncompound) structure change. The behavior of two systems differs by at most a small amount if they solve approximately the same class of problems, for example, can perform adequately in approximately the same set of environments. The appropriate criteria for two classes of problems to be approximately the same depends on the cost of failure. The weakest reasonable definition is that all systems which perform different but defined computations differ by a small amount as long as their computations terminate. We will say that two systems whose behavior differs from each other in this weak sense of small are weakly transformable into one another if they can be transformed into one another by a sequence of elementary structural changes. Weak transformability can be taken as a necessary condition for gradual transformability.*

In order for a system to be evolvable it must be capable of accepting at least one structural change. A system accepts a structural change if its performance improves or if it is capable of lasting long enough to accept another change, eventually leading to an improvement. A system remains evolvable as long as it satisfies this threshold condition. The reasoning here is that the probability of undergoing a transition to an acceptable structural form becomes negligibly small if simultaneous structural changes (e.g., mutations) are required. If p is the probability of a structural change, the time required for evolution scales as $1/Ap^m$, where m is the number of simultaneous changes that must occur and A is the number of systems in the population. By contrast the time required for evolution to proceed through m single step mutations scales as m/Ap. The evolution time becomes unacceptably large for all values of $m > 1$.

For example, if $p = 10^{-8}$, which is actually large for a biological mutation rate, the rate of evolution would be 10^8 times slower when $m = 2$ than when $m = 1$ for at least one of the possible mutations. Large values of p are not reasonable since this increases the chance of unfavorable structural changes canceling out favorable ones. Detailed calculations which take into account the effects of such canceling mutations, the relative advantage of improved forms, and the number of single step mutations required have been presented elsewhere (Conrad *1983*). These factors, and also the population size, have a negligible impact on the overall picture. Evolution is unfeasible

on any reasonable time scale if the evolutionary system is structured in such a way that double or simultaneous structural changes are required for the appearance of an improved form.

The threshold condition can be put into a neat form, analogous to a threshold condition for evolution formulated by Maynard-Smith *1970*. Let N denote the number of single event structural alterations that a system can undergo. In proteins N denotes the number of single mutations or single crossover events. In a computer program N denotes the number of single alterations of code, at any level of hierarchical organization. Let f denote the fraction of these alterations that are acceptable. In order for a system to evolve it must satisfy the threshold condition $fN \geq 1$.

A slight alteration in a system is more likely to be acceptable if it leads to a gradual modification in its behavior. Radically modified behavior might or might not be acceptable, whereas gradually modified behavior is intrinsically acceptable since it entails only small improvement or degradation of performance. Whether gradual transformability is a necessary and sufficient condition for an evolutionarily flexible system to be evolvable or just a sufficient condition depends on the definition of smallness and on the selection pressures imposed. It is possible, though highly implausible, that in some cases similarly structured systems with radically different behaviors could perform satisfactorily in the same environment. However, it can hardly be expected that this will be the case if the difference between the systems is so great that one gives a defined computation and the other does not. Weak transformability can thus be taken as a necessary condition for evolutionary adaptability under the reasonable assumption that mutations from defined to undefined computations are unacceptable.

By itself, however, gradualism does not assure evolutionary adaptability. The gradual transformability must be capable of generating a variety of useful forms. Thus evolutionary adaptability also depends on evolutionary flexibility.

We can summarize the necessary and sufficient conditions for evolutionary adaptability in the following:

Lemma 4.1 (conditions for evolutionary adaptability): *Evolutionary flexibility is a necessary condition for evolutionary adaptability. Gradual transformability is a sufficient condition for an evolutionarily flexible system to be evolutionarily adaptable, and weak gradual transformability is a necessary condition. Gradual transformability increases sharply when $fN \geq 1$, where N is the number of single event structural alterations and f is the fraction of acceptable alterations. It increases moderately with increases in f that push the system further above this threshold condition and decreases sharply when the system falls below this threshold condition.*

Structurally nonprogrammable systems have much higher evolutionary flexibility than structurally programmable systems, assuming that the structural alterations of the latter are constrained to be compatible with structural programmability. The evolutionary flexibility could nevertheless be adequate for performing a wide variety of tasks if the problem of discovering the allowable variants through an evolution process is ignored. Gradual transformability is essential for such discovery, however.

The gradualism property is most likely to be present in systems if they are highly parameterized and highly redundant. It is usually possible to change the parameters of a program over some range without radically altering the execution sequence. The incorporation of continuous dynamics increases parameterization since this increases a system's amenability to continuous deformation. A mouse and an elephant probably share the same rule of development, so far as the overall order of events is concerned, but differ by stretching and compression of events. Gradualism is also increased if it is possible to insert or delete rules that are independent so far as the operation of other rules are concerned, as in a production system type knowledge base of an expert system.

Redundancy serves to buffer the effect of structural change on behavior. Redundancy, taken by itself, confers fault tolerance rather than transformability (Dal Cin *1979*). However, it can serve to enhance transformability in the presence of parameterization and continuous dynamical features. Redundant, noncritical components can serve to absorb some of the impact of a structural change, thereby modulating its effect on components critical to the behavior of the system (Conrad *1983*).

Evolutionary systems can be associated with an adaptive surface which assigns a performance measure to each state of the system. If an evolutionary system fails to satisfy the threshold condition $fN \geq 1$, it becomes trapped on a particular peak on this surface. A system comprising a given number of particles could never be organized to definitely exclude such trapping since whether or not a mutation leads to a small change in behavior depends on the pressures of selection, which in turn changes as the organization of the system changes. However, redundant features can always be added in a stepwise manner without significantly degrading the performance of the system. Mutations of this type serve as f-enhancing mutations. The time required to reach a higher adaptive peak through a series of u f-enhancing mutations followed by m fitness-increasing mutations scales as $(u+m)/Ap$, which is very much faster than $1/Ap^m$, the time required for a double mutation, even when u is large. In an evolutionary system such f-enhancing mutations can hitchhike along with the fitness increasing traits whose evolution they facilitate. The addition of redundancy increases the dimensionality of the space and by doing so opens up *extradimensional* bypasses to higher adaptive peaks (Conrad *1979*).

5. *Programmability and Evolvability*

We now show that structurally programmable systems do not in general meet the requirement for gradual transformability.

Theorem 5.1 (gradual transformability problem): *The problem of ascertaining whether a structurally programmable system will undergo a small change in behavior in response to either single or multiple changes in its structure is unsolvable. It is in general impossible to put any reasonable metric on the amount of behavioral change that is likely to result from a structural change.*

The proof is similar to that of the unsolvability of the halting problem (Turing *1936-7*). Suppose that it is in fact possible to write a program, U, to solve the gradual transformability problem. It is sufficient (as in Definition 4.1) to take the behavior of two systems, P and P', as differing by at most a gradual transformation if they both give defined computations (that is, eventually come to a halt state) and as differing by an unacceptably large amount if one system gives a defined computation and the other does not. We are free to take for P any program that halts. The program U must go to a halt state at least when P' does not. We would then certainly get an answer to the gradual transformability problem since P' would halt if it is similar to P and U would halt if it is not. Suppose that U is itself P'. Then U halts if it does not halt. Since this is a contradiction, the assumption that U is a possible program must be incorrect.

Note that the gradual transformability problem is unsolvable even for the weakest sense of transformability. If it is not possible to ascertain whether an altered system will differ by an infinite amount (as when its computation becomes undefined) it is certainly impossible to solve it when the systems differ by a smaller amount. This is why no reasonable metric can in general be put on the expected amount of structural change. As a consequence structurally programmable systems in general fail to meet both a necessary and a sufficient condition for evolutionary adaptability. For finite systems the notion of unsolvability must be replaced by that of intractability. In the absence of specializing assumptions, it is as intractable as proving program correctness. Experience with digital computers in fact suggests that the vast majority of arbitrary changes in rules lead to undefined or useless behavior.

The unsolvability (or intractability) of the gradual transformability problem in general does not mean that it is unsolvable in all particular cases. We have already observed that rules displaying features of continuity and

states displaying features of redundancy allow for gradual deformation of the execution sequence (or, alternatively, realization of a given sequence by a deformed organization). The presence of continuity is incompatible with structural programmability. This is why the unsolvability of the gradual transformability problem does not preclude a reasonable metric for behavior change in structurally nonprogrammable systems. In many instances it is possible to ensure that a structurally nonprogrammable system undergoes acceptably small behavior change by "mutating" it into a higher dimensional space.

Evolution processes in nature have produced a wide variety of powerful biological information processing systems. According to the strong form of the Turing-Church thesis it should be possible to simulate these evolution processes with structurally programmable computers. That is, it should be possible to use structurally programmable computers to build virtual machines capable of supporting evolution. To do so it is necessary to invest sufficient computational resources to pay for the cost of simulating the dynamical features and redundancies that support evolution. If it is not in principle possible to simulate these mechanisms then it must be admitted that an important class of problem solving processes in nature, namely evolution processes, are not simulatable by digital computer.

The computational efficiency of a structurally programmable system is in effect reduced if some of its resources are used to simulate evolution-facilitating features. To properly formulate the relation between evolutionary adaptability and structural programmability we therefore require the following

Definition 5.1 (effective computational efficiency): *A virtual machine is a simulation built on top of a structurally programmable base machine. The effective number of interactions that a virtual machine uses for computing is the number of interactions that the base machine uses for computing minus the number of interactions used for the simulation. The effective computational efficiency of a virtual machine is the effective number of interactions it uses for computing divided by the maximum number possible in a system consisting of the same number of particles as the base machine.*

Theorem 5.1 and Lemma 4.1 combined with this definition may be expressed as

Theorem 5.2 (structural programmability vs. evolutionary adaptability): *Structurally programmable systems fail to satisfy the conditions for evolutionary adaptability when the mutations of structure occur at the level of the base machine. Structurally programmable systems can be used to achieve evolutionary adaptability through simulation, in which case the structural mutations occur at the level of the simulation. Such simulated evolutionary adaptability can only be achieved at the cost of a decrease in effective computational efficiency.*

6. The Trade-off Principle

The results obtained in the preceding sections can be summarized in

Theorem 6.1 (trade-off principle): *A computing system cannot have all of the following three properties: structural programmability, high computational efficiency, and high evolutionary adaptability. Structural programmability and high computational efficiency are always mutually exclusive. Structural programmability and evolutionary adaptability are mutually exclusive in the region of maximum effective computational efficiency (always less than or equal to the computational efficiency).*

We may also gather together a number of more specific results:

1. The potential computational efficiency of a structurally programmable system is less than that of a structurally nonprogrammable system, the former decreasing with the number of particles (or processors) and the latter being independent of the number of particles. The potential computational power of a structurally programmable system is a constant that depends on the size of the user's manual (that is, on the length of the definitions of processors comprising the machine), whereas the potential computational power of structurally nonprogrammable systems increases as the square of the number of particles. Furthermore, if a structurally programmable system is operated in a sequential (truly programmable) mode its computational power decreases as the number of particles increases (Theorem 3.1).
2. The potential evolutionary adaptability of structurally programmable systems is less than that of structurally nonprogrammable systems, the former having less potential evolutionary flexibility and failing to meet the requirement for gradual transformability at the level of the base machine (Lemma 4.1 and Theorems 3.1 and 5.2).
3. Virtual evolutionary adaptability can be exhibited by structurally programmable systems if these systems allocate enough of their time and processor resources to simulating the structure-behavior relations that allow for gradual transformability and evolutionary flexibility. This allocation of computational resources reduces effective computational efficiency (Theorem 5.2 and surrounding discussion).

The trade-off principle does not exclude the coexistence of high evolutionary adaptability and high computational efficiency. In general evolutionary adaptability and high computational efficiency would go together since an evolvable system could learn to use its resources efficiently. This claim could only be satisfactorily demonstrated by construction and demonstration, through computer simulation, that the desired properties are indeed achieved (see next section). However, it is likely that biological organisms provide an instantiation of this claim. As products of evolution, organisms cannot be structurally programmable. Furthermore, they appear to be highly effective for certain types of computations as compared to structurally programmable machines. Similarly, point 3 above has not been given either a constructive or analytical interpretation. However, it is possible to give a fairly general analysis of the organizational features that provide an effective substrate for evolution (Conrad *1983*) and there is nothing about these features which precludes simulation. This does not establish that evolvability comparable to that exhibited by biological systems can be achieved through simulation. It is probably impossible to definitively establish such a claim. However, if the strong form of the Turing-Church thesis is taken as axiomatic it is then possible to assert the simulatability of evolution given sufficient computational resources, or alternatively, to assert that the nonsimulatability of evolution would invalidate the strong form of Turing-Church.

7. *Evolutionary Machines*

Evolutionary and genetic algorithms have been used for optimization problems (Bremermann *1962,* Holland *1975*) and for adaptive pattern recognition (Conrad et al. *1987*). The trade-off principle implies that such algorithms should be conceived as consisting of two parts. The first is the variation and selection procedure per se. The second is the structure of the substrate on which the search procedure acts. This determines the structure of the adaptive surface on which the evolution is taking place. For evolutionary methods to be effective the substrate should have the evolution facilitating properties of gradualism and flexibility.

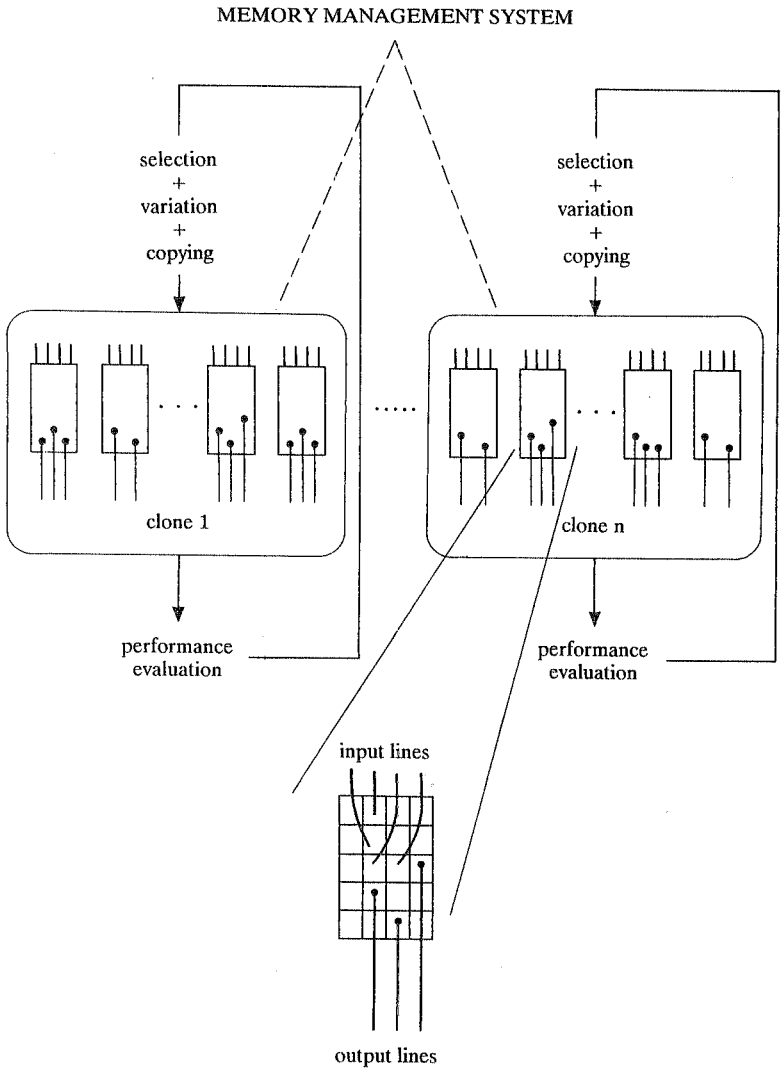
The arguments leading to the trade-off principle suggest that it may be possible to advantageously use structurally programmable systems to simulate these evolution-facilitating features. Many of the resources of structurally programmable systems are inefficiently recruited for computing when these systems are run in a programmable mode. *Representing evolution-enhancing features in structurally programmable systems can finance itself to*

the extent that it increases the effectiveness with which computing resources are used for problem solving.

Consider first virtual implementations in which the base machine runs in a sequential mode. The computational effort required to simulate an evolution-facilitating substrate entails a decrease in effective computational efficiency. However, this produces a distinctive evolutionary learning capability that may open up problem domains that could not be as effectively addressed by direct use of the base machine. Problems whose solution demands an adaptive component fall into this domain. Effective programmability is given up at the level of the virtual machine, just as it is in highly parallel machines. As a consequence such machines lend themselves to implementations in which the simulation of the dynamical processes required for evolution are run on many processors in parallel. This simulation, though costly, can be financed by the gain in computational efficiency that results from tapping otherwise dormant processors.

The architecture depicted in Figure 1 illustrates how this could work. The basic modules are cellular automata with input lines and read heads. Each cellular automaton is an array of subcells, each of which is in one of a number of possible states whose transitions are influenced only by neighboring subcells. Such arrays are capable of exhibiting highly elaborate patterns of behavior, ranging from any type of pattern that can be exhibited by partial differential equations to patterns so elaborate that they cannot be predicted in advance except through step-by-step enumeration (Wolfram *1986*). The local rules can be viewed as serving to integrate patterns of input signals in space and time. Particular subcells will thus be activated at particular times in response to different families of patterns. Some of the subcells contain read heads, while others do not. If a read head is located on a subcell that is activated it produces an output signal.

A collection of individual modules with the same cellular automaton dynamics will be called a clone. The clone is the next higher level of organization in the machine. The variation and selection algorithm acts randomly, adding or deleting read heads on each member of the clone and evaluating how well it performs a desired transformation of a training set of input signal patterns to output signal patterns. The configuration of read heads in the best performing module is copied into the other members of the clone, and the process is repeated until the training set is learned. If the local cellular automaton dynamics are such that the training set cannot be learned, then the variation and selection procedure must be applied to the cellular automaton dynamics itself. In this way a variety of differently adapted clones with the same cellular automaton dynamics are harvested after a first round of learning cycles, while clones with different cellular automaton dynamics are harvested after a second round of learning cycles.



If the cellular automaton dynamics causes different input signal patterns to activate the same subcell, it serves to aggregate them into an equivalence class. This is why the machine can generalize from a small training set of signal patterns. However, if such aggregation is admitted it is always possible to define tasks that the dynamics are incapable of performing; this is why the higher level of evolution on the dynamics is necessary. Also note that the cellular automaton dynamics can be chosen so as to be more or less structurally stable (since partial differential equations can be computed via cellular automata). As the dynamics becomes structurally more stable, it becomes more gradually transformable. The cellular automaton modules thus provide both gradual transformability and evolutionary flexibility, the two properties required of a substrate suitable for evolutionary learning.

The top layer of the machine is a memory management system that puts together combinations of differently adapted modules for the performance of complex tasks. Each memory control unit, or reference module, is connected to each of the cellular automaton modules. The activation of a reference module activates a cellular automaton module if its connection to it is facilitated for signal flow. Thus each reference module controls a combination of cellular automaton modules. This organization allows for a higher level of evolutionary learning. However, it is not necessary to describe the operations of this memory level of organization here (see Conrad *1976*).

We have built simulation systems that operate according to the above principles (using modules with differential equation dynamics) and have reported on these elsewhere (Kampfer and Conrad *1983*; Kirby and Conrad *1984, 1986*; Conrad et al. *1987a*). Efficient, hardwired systems for simulating cellular automaton dynamics (cf. Toffoli *1984*) would allow for the efficient utilization of low level parallelism. Such systems are still structurally programmable since the local rules can be effectively prescribed, though in practice it is not possible to prescribe a desired global rule. Continuous analogs of cellular automaton dynamics would be truly structurally nonprogrammable. Biological organisms consisting of molecular components are truly nonprogrammable since it is not in general possible to prescribe desired local rules by means of structural mutations. Nevertheless by linking a structurally nonprogrammable collection of pattern processors together in a suitable way, it is possible to duplicate any computation that could be performed by a structurally programmable machine (Conrad *1974a*). The potential efficiency is greater since each processor can be specifically tailored for the task it performs.

Architectures of the type described could be efficiently implemented using modifications of current silicon technology. The miniaturization of conventional chips is limited by physical side effects, in particular quantum mechanical tunneling of electrons. Tunneling, however, could be exploited

to achieve the type of neighbor-neighbor interactions that are necessary for cellular automaton dynamics. Electron diffusion, another phenomenon that must be suppressed in a chip engineered to realize conventional logic, is an excellent type of dynamics for modules in an evolutionary architecture. Macromolecules of the type that occur in organisms would undoubtedly provide the best substrate for evolutionary computing, though. This is due to the fact that molecules such as proteins have highly specific readout capabilities and in addition a high degree of evolutionary gradualism and flexibility.

8. Conclusion

The trade-off principle establishes a link between computing and evolution. Algorithms can be executed in structurally programmable or structurally nonprogrammable systems. The potential computational efficiency is greater in the latter case. Traditional programming, or prescriptive control over the rule obeyed, is given up. However, such systems are well suited for programming by evolutionary means. Evolution is itself a method of problem solving. This form of problem solving can be expressed in an algorithmic framework, but it is different from the algorithms usually implemented on computers in a significant respect. Evolutionary programming involves both a search procedure and a substrate on which this procedure acts. The search procedure can be expressed as an algorithm in a straightforward manner. The substrate can also be described in algorithmic terms, by means of simulation. However, the characterization of the substrate is of such immense importance for the effectiveness of evolution and so closely connected with choice of hardware, that it is probably useful to place evolutionary problem solving in a distinct category.

The recognition of the importance of substrate architecture for evolutionary computing opens up the possibilities of new classes of machines, both virtual and real, that in fundamental respects are more biology-like than present day machines. This class includes machines programmed to simulate the dynamical processes and redundancies that facilitate evolution and, as well, machines directly structured to display these features. The analysis suggests that for some problems the cost of simulating the evolution-enhancing features that make biological materials a good substrate for evolution may be outweighed by the resulting enhancement of adaptive capability and by the more effective use of parallelism.

We can finally note that the trade-off principle has a fundamental epistemic implication. Computer scientists commonly assume that it is possible to simulate any mathematically describable process in nature with digital computers, provided that enough time and processor resources are available. It is impossible to prove this strong version of the Turing-Church thesis; but up to the present time there has been no convincing counterexample. On the surface Turing-Church suggests that our understanding of a process, such as a process of human intelligence, can always be crystallized in a computer program and then communicated to an actual machine for execution. The tacit assumption is that this programmability would not itself preclude the computational efficiency necessary for simulating the process in nature. This assumption is incompatible with the trade-off principle. Many processes in nature must be such that we cannot understand them in terms of a computer program and at the same time put our understanding to the test by running the program on a machine. Brain processes of intelligence fall into this category, since the brain is a product of evolution and thus cannot be structurally programmable. Conceivably we could evolve an artificial system to simulate brain-like intelligence; but we would then find it just as difficult to specify and test the program of this artificial system as to specify and test the program that generates the behavior of an organism.

References

Arbib, M.A.
1969 *Theories of Abstract Automata* Englewood Cliffs, NJ: Prentice-Hall (1969).

Avizienis, A.
1983 Framework for a taxonomy of fault tolerance attributes in computer systems. In: *IEEE Conference Proceedings of the 10th Annual International Symposium on Computer Architecture,* pp. 16–21 (1983).

Conrad, M.
1972 Information processing in molecular systems. *Curr. M. Bio.* (now *Biosystems*) **5/1** (1972) 1–14.
1974 Molecular automata. In: *Physics and Mathematics of the Nervous System,* eds. M. Conrad, W. Güttinger, and M. Dal Cin. Heidelberg: Springer-Verlag (1974) 419–430.
1974a Molecular information processing in the central nervous system, part I: Selection circuits in the brain. In *Physics and Mathematics of the Nervous System,* eds. M. Conrad, W. Güttinger, and M. Dal Cin, pp. 82–107. Heidelberg: Springer-Verlag (1974).
1976 Molecular information structures in the brain. *J. Neurosci.* **2** (1976) 233–254.
1979 Bootstrapping on the adaptive landscape. *Biosystems* **11/2, 3** (1979) 167–180.
1983 *Adaptability.* New York: Plenum Press (1983).
1984 Microscopic-macroscopic interface in biological information processing. *Biosystems* **16** (1984) 345–363.

1985 On design principles for a molecular computer. *Comm. ACM* **28** (1985) 464–480.

Conrad, M., and H.M. Hastings

1985 Scale change and the emergence of information processing primitives. *J. Theor. Bio.* **112** (1985) 741-755.

Conrad, M., R. Kampfner, and K.B. Kirby

1987 Simulation of a reaction-diffusion neuron which learns to recognize events (Appendix to: M. Conrad, Rapprochement of artificial intelligence and dynamics). *Eur. J. Oper.* **30** (1987) 280–290.

1987a Neuronal dynamics and evolutionary learning. To appear in: *Advances in Cognition: Steps Toward Convergence,* eds. M. Kochen and H. Hastings. Washington, D.C.: American Association for the Advancement of Science (1986).

Dal Cin, M.

1979 *Fehlertolerante Systeme.* Stuttgart: Teubner Studienbücher (1979).

Garey, M., and D. Johnson

1979 *Computers and Intractability.* New York: Freeman (1979).

Harary, F., and E.M. Palmer

1973 *Graphical Enumeration.* New York: Academic Press (1973).

Holland, J.H.

1975 *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI: University of Michigan Press (1975).

Kampfner, R., and M. Conrad

1983 Computational modeling of evolutionary processes in the brain. *B. Math. Biol.* **45** (1983) 931–968.

Khinchin, A.I.

1957 *Mathematical Foundations of Information Theory.* New York: Dover (1957).

Kirby, K.G., and M. Conrad

1984 The enzymatic neuron as a reaction-diffusion network of cyclic nucleotides. *B. Math. Biol.* **46** (1984) 765–783.

1986 Intraneuronal dynamics as a substrate for evolutionary learning. *Physica* **22D** (1986) 205–215.

Kuck, D.

1978 *The Structure of Computers and Computations,* vol. 1. New York: John Wiley and Sons (1978).

Maynard-Smith, J.

1970 Natural selection and the concept of a protein space. *Nature* **225** (1970) 563–564.

Pattee, H.H.

1970 Physical problems of decision-making constraints. In: *Physical Principles of Neuronal and Organismic Behavior,* eds. M. Conrad and M. Magar, pp. 217–225. New York: Gordon and Breach Science Publishers (1970).

Toffoli, T.

1984 CAM: A high-performance cellular automaton machine. *Physica* **10D** (1984) 195–204.

Turing, A.M.

1936-7 On computable numbers, with an application to the Entscheidungsproblem. *P. Lond. Math. Soc. (2)* **42** (1936-7) 230–265.

Wolfram, S.

1986 Approaches to complexity engineering. *Physica* **22D** (1986) 385–399.

The Universal Turing Machine
A Half-Century Survey

edited by

Rolf Herken

Institut für Theorie der Elementarteilchen
Freie Universität Berlin

OXFORD UNIVERSITY PRESS
1988