

IMPACT OF NON-CARTESIANISM ON SOFTWARE ENGINEERING

TAKASHI GOMI

Applied AI Systems, Inc.

Suite 600, Gateway Business Park

340 March Road, Kanata, Ontario Canada K2K 2E4

e-mail: gomi@Applied-AI.com

This paper proposes a novel way of looking at the concept of programs and programming by focusing on a recent development in the method of implementing the process structures necessary to operate intelligent robots and describing its characteristics in the context of Software Engineering. While still proven effective only in a new breed of robotics, there is a possibility the methodology is applicable to a wider range of embedded computing, realtime processes, and potentially to some parts of information processing, particularly if it is combined with evolutionary computational techniques such as GA. The approach could potentially become a crucial programming paradigm, forcing a new way of looking at the very concept of programming.

1 Introduction

The rise of Structured Programming^{1,2} meant structuring the components of a program based on an organizational principle, a better encapsulation of program modules, and enforcement of coding disciplines on each of the modules. It also meant the introduction of new concepts in the manner the structure gets implemented. For example, Yourdon emphasized the importance of having all essential modules of a system to be developed identified and registered at the outset. This was encouraged even when it was not clear what their actual algorithm would be. In short, Software Engineering was a serious attempt to bring discipline into the theretofore black art of programming. Although programming still is largely a black art, the approach has been successful in the greater software engineering community and allowed practitioners of programming both in academia and industry easier comprehension of program structures and a more controlled manner of implementing and analyzing algorithms. Structured Programming also established a methodology to deal with the concepts of *module strength* and *module decoupling*, and hence significantly increased the reliability of software in general, and resulted in more efficient production of software. Object-Oriented programming, and some of the *agent* system concepts are the ultimate manner of expressing this approach to running computers. A computer is viewed as a form of automaton where every entity in the system is made accountable for its effects and every state transition traceable.

The move toward a more well-disciplined approach to running computers, however,

did not solve the problem of rigidity of computation, and the problems associated with the need to have everything explicitly defined and instructed. Every bit in a program structure needed to be defined in its relationship to a global structure and its values accountable at all times. Computerized systems must be *perfect* at all times and failure to maintain these premises is invariably linked to a system failure. A screen of a personal computer that freezes, a banking network which can crash unexpectedly or an aircraft that can experience a sudden seizure on one of its control surfaces are just a few examples of such a failure. Since it is now common knowledge that total validation of even a moderately complex computerized system is impossible, this is particularly a serious drawback in realtime systems and embedded computing where a large risk, including potential loss of life, is involved. One can see this rigidity as a result of pursuing the Cartesian view on systems and machines. In Cartesian doctrine, which is in the backbone of traditional science, everything needs to be defined, every transition must be proven, and every event needs to be explained.

The notion of *soft computing* being pursued by a number of research groups today is obviously meant to ease the harshness we face by having to depend on the *perfectionism* insisted upon by computers. However, the movement still seems to center on the idea of easing up the restrictions of the hard and rigid framework of conventional computation using tools such as neural networks and fuzzy set theory. However, what is being sought now in terms of softening will not be satisfied by these “padding techniques”. The softness must come from the architecture itself and the philosophy that underwrites the architecture. It is us humans who assumed machines and systems need to be *controlled* to the minutest detail and all their behaviors accountable to humans. A system is broken down, typically by its function, into major elements. Each of the elements is then broken down into smaller elements. The process is repeated recursively until all bits and their relationships to others in the system are explained and documented. This process of ‘divide and conquer’ is not only extremely costly, but also immutable in that once concepts, events, object, or ideas are defined any changes involve a considerable amount of time and effort on the part of the designer. Systems thus produced also lack in flexibility in operation and often cannot adapt to changes that normally happen in the environment where the system is put into use. They are hard to maintain and difficult to comprehend, often even by those who created the systems. Systems designed using the conventional concept of *control* are *closed*. The idea of *autonomy* is a key to get out of this bind and build systems that are *open*.

2 Non-Cartesian programming

2.1 Background

The 1980s saw the rise of an approach to tackle the issue of artificial intelligence (AI) from a new angle. Rodney Brooks of The Massachusetts Institute of Technology (MIT) proposed Subsumption Architecture in 1986³ as a very unique, and controversial way of running intelligent robots. His approach is based on the principle of self-organization that results from dynamic activities of simple agents. This is not too different from what Stephanie Forrest proposed as a concept in the early 1990s as *emergent computation*⁴. Around that time the greater scientific community itself was also opening its eyes to the new manner of looking at the world and the reality as a *complex system* partly through a new discipline of research called Artificial Life⁵.

As evident in most of the computerized systems in use and under development today, the whole purpose of creating a computer program is so that automation with some level of complexity and sophistication could be realized the way the instigator of the system and ultimately the programmer have wanted. In other words, no matter how complex and detailed the program appears, a computer is no more than an elaborate automaton that executes the intended algorithm. All steps executed are accountable and explainable. Necessary intelligence is said to be generated out of the explicit execution of known steps.

The non-Cartesian view on intelligence differs from that implied in conventional programming methodologies in a simple but fundamental manner. In his discussion on the nature of intelligence Brooks states, "Intelligence resides in the eyes of the beholder"⁶. Intelligence, as beauty, is viewed as a property detectable (visible, audible, touchable, etc.) only in a context, and in relation to an observer, and not as an independent entity by itself. Development of a system based on functional decomposition (the dominant design approach today) clearly ignores this view. Many researchers also misunderstood what intelligence really is, as eloquently pointed out by Rolf Pfeifer as "the frame of reference problem".⁹ The failure to recognize this shift in epistemological stance on the nature of intelligence was one of the mistakes made by researchers in traditional AI for the past few decades. Table 2.1 summarizes a number of key AI researchers and roboticists who share the new view on intelligence or the approach in building systems.

Table 2.1 Key researchers active in "New AI"

Stephanie Forrest (Professor, U. New Mexico)	<i>Emergent Computation</i>
--	-----------------------------

Pattie Maes (Associate Professor, MIT Media Laboratory)	<i>Action Selection Dynamics (ASD)</i> , Agent-based Approach, Soft Robots, Interface Agents
Rodney Brooks (Professor/Director, MIT AI Lab)	<i>Subsumption Architecture (SA)</i> , Behavior-Based AI
Chris Langton (Director Artificial Life Program, Sante Fe Institute)	<i>Collectionism</i>
Rolf Pfeifer (Professor, Head, Artificial Intelligence Lab, U. Zurich)	‘Fungus Eater’, <i>New AI</i> , non-Cartesian computation
Luc Steels (Director, Sony Computer Science Laboratory, Paris)	<i>Selectionism</i>
Thomas Christeller Director, German National Research Center for Information Technology	Social and Reflective Robots, Software agents
Ian Horswill (Assistant Professor, Northwestern University)	Habitat Constraint Computational Vision, non-Newtonian computation
Jean-Arcady Meyer Ecole Normale Supérieure	<i>Animat</i> Approach and <i>Biorobotics</i>
Inman Harvey (Research Fellow, Evolutionary Robotics Group, U. Sussex)	<i>SAGA</i> paradigm
Stewart Wilson (Roland Institute)	<i>Animat</i> Approach

2.2 What is non-Cartesian programming?

The basic unit of a non-Cartesian program is a simple expression of causality, such as a production rule consisting of an <if ... then ...> sentence. As explained further in Section 3 below, it can be implemented in a number of ways including hardware implementation. In non-Cartesian programming, an agent or the expression of causality, however

implemented, is arranged in such a manner that it can be invoked, totally independent of other agents, by some external input(s) when a condition to invoke it is satisfied. In this regard, non-Cartesian programming is not just another agent system. There is no main routine to call an agent, no caller/callee relationships between an agent and another program body, no centralized scheduling of tasks that invoke agents, nor centralized control mechanism in the case of hardware implementation. And there is no hierarchical arrangement of the units, components, or agents in terms of the levels of abstraction which they represent or at which they operate. The agents are simply laid out side by side, each one of them independently *invokable* asynchronously and at its own initiative. In short, the system implemented in non-Cartesian programming is fully distributed in its structure, without a control mechanism, and its operation is autonomous. The lack of center of control is particularly important. A process can be invoked totally by an external condition perceived by an agent and does not require another entity to exert a control for it to be executed. The external condition for invocation could be any form of sensor inputs, signaling (reporting) by a human through an input device, or the arrival of a message via a communication channel of any sort.

The result of agents thus invoked is a process structure that emerges temporarily with its own dynamic and tentative control hierarchy and inter-process relationships as in the 'bottom invoked top structure' in Langton's *Collectionism*. The structure is only dynamically supportable and, even in this form, in most cases cannot be sustained permanently due to changes that naturally occur in its own operational environment. The structure is not meant to execute an algorithm either. An algorithm to represent functioning of such a dynamic structure does not exist *a priori* and the simple causality that is described within each agent is too simple and minute to be termed an algorithm. The entire organization is a form of *complex system* in the sense that the total effect of the system is often greater than the sum of the effects of each process involved. In contrast, conventional computation based on algorithms is but a deterministic *simple system* no matter how complex the algorithms and/or their implementation could be. The designer of the program knows explicitly what is to be achieved and the programmer simply documents the process to cause the desired set of effects.

The conventional approach is also Cartesian as the algorithms involved are sought so as to establish computability of a proposed solution to a perceived problem. The entire scheme fits nicely in the Cartesian framework of deductively proving a hypothesis, as discussed in great detail by Descartes⁸. Indeed, conventional programming is Cartesian in its spirit and practice. It is a process of deductively proving an algorithms through top-down breakdown of functions, inputs, and outputs, and documenting the broken-down fragments until the last bit is documented. The proposed non-Cartesian scheme differs drastically from this approach, and works well at least when applied to intelligent robots which try to achieve practical tasks in a real environment. Such an implementation in the form of a robot is said to be *situated* and *embodied*. It is still not clear and at best

debatable if and how a non-Cartesian program performs in conventional information processing which is *unsituated*, *unembodied*, or both.

2.3 An example non-Cartesian program

(1) A simple program to run a robot in a complex environment

Figure 2.1 shows the agent structure of an example non-Cartesian program implemented for a behavior-based robot called R2, and Figure 2.2 presents the actual code that was written to implement the agents and the structure. The robot's hardware was originally produced by Brooks' group at MIT. The experiment was designed by the author and the program was implemented by J-C Laurence⁹ of Applied AI Systems, Inc (AAI) in the form of Subsumption Architecture. It describes agents collectively intended to give R2 the ability to run through a narrow passage, such as the one shown in Figure 2.3, avoiding contact with the walls, a bottleneck in the passageway, and later, another moving robot. Narrow corridors and passageways of this type are often found in factories and warehouses. In fact, the experiment was meant to test a prototype mobile platform for investigating effectiveness of behavior-based techniques in transportation applications such as AGV (Autonomous Ground Vehicle). An analysis of the tasks involved had been conducted and a set of component behaviors or agents to synthetically generate the behaviors of the robot in *all* circumstances anticipated in the set up was determined.

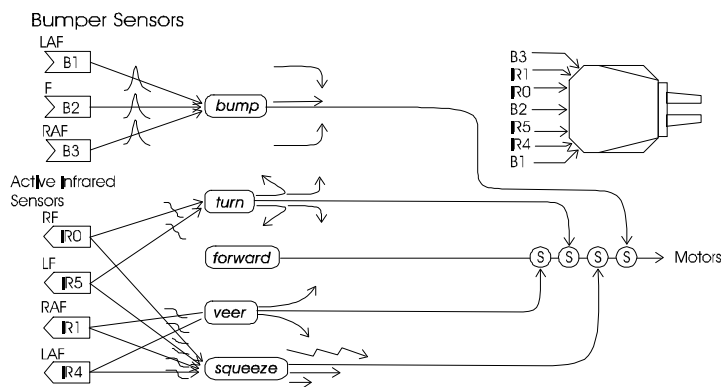


Figure 2.1 Agent structure of a program that allows passing of intelligent robots through a narrow passage.

The behavior set is described as five independent agents that are not linked directly,

but only through the environment via sensor inputs and output currents sent to two motors. The R2 robot, in this experiment, has 3 bump sensors, 4 active infrared sensors, and two motors. Left and right front wheels are differentially driven by these motors, while the rear of the robot is supported by two omni-directional casters. Other sensors which surround the circumference of the robot are ignored in defining the behaviors as they do not significantly contribute to the motions of the robot in this experiment. Each of the bump sensors generates an electronic pulse which manifests itself as a non-zero integer value in a register after an analog-to-digital (AD) conversion.

Each of the active infrared sensors emits an infrared signal at a specified duty cycle (at about 30 Hz) and returns an analog value on the receptor segment of the sensor. The value becomes the strength of the reflected signal as an integer in a register. These register values are then directly converted by a corresponding agent into a pair of values that designate the size and polarity of current fed to the motors using pulse width modulation (PWM). The actual energy sent to the motors is determined by associated servo controller circuitry.

The important aspect of this short example program is that it only defines a set of simple actions that the agents would take if invoked. Despite their simplicity, upon invocation, they collectively generate sufficient intelligence to drive the robot through the passageway and around any obstacles, including another mobile robot. The agents do not contain a description of how the robot avoids the walls, bottlenecks in the path, and other robots, or for that matter, how and in which direction it proceeds in the corridor in the first place. It only specifies at a very low level how and when the motors react in response to specific sensor inputs. The intelligence is generated through the phenomenon called *emergence*¹⁰ as it is known in the study of *complex systems*. *Emergence* results from an asynchronous invocation of dynamic and non-linear processes. Although emergence is caused by a number of parallel agents in which the

```
(include "plsys:plutils.beh")
(include "r2sys:r2control.beh")

(defconstant $straight 0)
(defconstant $straight-hi 11)
(defconstant $left-turn 3)
(defconstant $right-turn 4)
(defconstant $left-veer 1)
(defconstant $right-veer 2)
(defconstant $halt 5)

(fingers-open)
(delay 3.0)
(finger-brake-on)
(lift-brake-off)
(lift-down)
(delay 1.0)
(lift-up)
(delay 1.0)
(lift-brake-on)
(output start 1)))

(defbehavior feet
:inputs (direction start)
:decls ((begin:init 0))
:processes(
  (whenever (received? start)
    (setf begin 1))
  (whenever (received? direction)
    (if (= begin 1)
      (sequence
        (cond
```

```

(= direction $straight) (move -22 -25))
(= direction $left-veer) (move -25 -12))
(=direction $right-veer) (move -12 -25))
(=direction $left-turn) (move -15 15))
(= direction $right-turn) (move 15 -15))
(= direction $halt) (move 20))
(=direction $straight-hi)(move -34))))))
(defbehavior bump
:outputs (direction)
:processes(
  (whenever (/= (bump 1) 0)
    (output direction $right-turn))
  (whenever (/= (bump 3) 0)
    (output direction $left-turn))
  (whenever (/= (bump 2) 0)
    (output direction $halt)))
)
(defbehavior turn
:outputs (direction)
:processes(
  (whenever (= (range 5) 4)
    (output direction $right-veer))
  (whenever (= (range 0) 4)
    (output direction $left-veer))
  (whenever (< (range 5) 4)
    (output direction $right-turn))
  (whenever (< (range 0) 3)
    (output direction $left-turn)))
)
(defbehavior forward
:outputs (direction)
:processes(
  (whenever t
    (output direction $straight)))
)
(defbehavior veer
:outputs (direction)
:processes(
  (whenever (< (range 4) 5)
    (output direction $right-veer))
  (whenever (< (range 1) 5)
    (output direction $left-veer)))
)
(defbehavior squeeze
:outputs (direction)
:processes(
  (whenever (and (< (range 1) 5)
    (< (range 4) 5))
    (> (range 0) 4)
    (> (range 5) 4)
    ;; speed.
    (output direction $straight-hi)))
  (whenever (or (and (= (range 4) 4)
    (> (range 5) 4)
    (> (range 0) 4)
    (> (range 1) 4))
    (and (= (range 1) 4)
    (> (range 5) 4)
    (> (range 0) 4)
    (> (range 4) 4)))
    (output direction $straight))
)

```

```

(whenever (and (< (range 0) 3)
  (< (range 1) 3))
  (output direction $left-turn)))
(connect (init start)(feet start))
(connect (default direction) (feet direction))
(connect (veer direction) (feet direction))
((inhibit (default direction)))
(connect (turn direction) (feet direction))
((inhibit (default direction))
  (inhibit (veer direction)))
(connect (squeeze direction) (feet direction) ;
  ((inhibit (turn direction)))
  ((inhibit (veer direction))
  (inhibit (default direction)))
(connect (bump direction) (feet direction))
((inhibit (default direction))
  (inhibit (veer direction))
  ; ((inhibit (turn direction)))
  ((inhibit (squeeze direction)))

```

Figure 2.2 Code that implements the agent structure of Figure 2.1

parameters have non-linear relationships among themselves, the emphasis is on *emergence*, **not** on *parallelism* per se.

In Figures 2.1 and 2.2, agent *bump* detects a contact with the environment through a set of contact sensors (B1, B2, B3) and generates a turn towards the opposite direction, or stops the robot if the collision is frontal. The *turn* agent prepares a pair of current values for the motors in response to the strength of reflection of two frontal infrared beams emitted by the active infrared sensors IR0 and IR5. The sharpness of the turn corresponds to the strength of the reflection and the direction of the turn corresponds to the direction of the reflection. The agent *forward* responds to a null sensor the value of which is always 'true' and issues a steady set of current values to drive the motors straight forward. The *veer* agent looks at the two front diagonal infrared sensors (IR1 and IR4) and generates a set of gradual opposite turn signals to the motors. This puts the robot into an oscillatory motion if two angular-front obstacles are located in a certain manner. The *squeeze* agent breaks the dead-lock by proceeding cautiously if there are obstacles front-diagonally (IR1 and IR4) but not frontally (IR0 and IR5). This allows the robot to 'sneak through' a narrow gap despite obstacle warnings from IR1 and IR4.

The outputs from these agents are put through a network to evaluate priorities. Each agent is given a priority at which its output is honored. The right-most output (output of the *bump* agent) in Figure 2.1 is given the highest priority while the left most output (of the *forward* agent) is treated at the lowest priority. The structure is implemented in the coding of Figure 2.2 using *connect* function.

(2) Adaptive trajectory of R2 robot

When the robot with the agents or component behaviors defined in Figures 2.1 and 2.2 is put into action in a narrow corridor illustrated in Figure 2.3, it proceeds forward (because *forward* process is always on) and maintains its course roughly straight in the approximate center of the corridor. This is because the process structure that emerges when *turn* and *veer* agents are executed tries to keep it so. If the R2 robot deviates from

this established norm for whatever reasons, these processes work together to bring it to this dynamic equilibrium. The robot proceeds while constantly adjusting its position, orientation, and speed, or its relationship in this equilibrium. The equilibrium in turn changes in accordance with situations at hand. An equilibrium a moment (say, 100 milliseconds) ago is more often than not different from the one that exists now. Since infrared reflections from the walls of the corridor are not uniform along the length of the corridor and the floor does not have an even spread of friction against the rubber-capped drive wheels of the robot, the exact trajectory of the robot is not always straight, rigid, nor precise just as trajectories of an animal would not be. This is particularly true when the robot is traveling through the narrow passage (the bottleneck) formed by the protruding structure (an odd-shaped box placed in the mock-up corridor) and a wall on the opposite side. The exact trajectory of the robot also varies from run to run for a number of reasons such as variation in starting position and/or orientation of the robot, difference in friction between parts of the rubber tire and the specific spots on the floor the robot happens to be traveling, and changes in ambient infrared levels and their pattern of fluctuations, etc. The *uncertainty* associated with the runs is due to the robot's dependence on its relationship to the operational environment as it is detected through the robot's sensors and experienced through its wheels and casters. Therefore, the runs are totally dependent on how the robot detects various facets of the real world and how

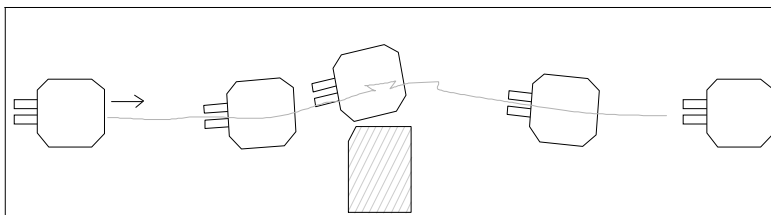


Figure 2.3 A narrow passage with an obstacle

it reacts to them. There is no 'theater of intelligence' which coordinates behaviors onboard the robot and manifests its authority by handing down the control to the drive wheels, as in most, if not all conventional mobile robots.

(3) The flexibility and the robustness of the robot's motion

To those accustomed to the concept of *control* as in control system theory where accuracy, rigidity, repeatability, exactness, assertiveness, and precision are a virtue, the volatile, non-deterministic, and fleeting nature of the robot's operation may seem

undesirable. However, this is the same set of characteristics fully autonomous beings such as insects, animals, and humans who are expert in dealing with an uncertain *open world* possess and use. The R2 robot described above is not an automaton under a strict direction but an autonomous machine. R2 in this set up, despite a very simple agent structure, can cope with a much wider range of situations than an ordinary *controlled* robot with identical hardware and a similar amount of software could. Among others, it can quickly respond to changes in its operational environment. If a new obstacle of arbitrary type is thrown into its path, the robot can, without prior knowledge of the location, shape, and dimensions of the obstacle, quickly determine in its action if it can safely pass by it or not, and if it can, through which trajectory and at what speed! If it can't initially, it repeats the search for a passage automatically and perpetually like an animal trapped in a corner. If someone lifts the robot up during a run and relocates it somewhere else in the corridor, or any other floor space, it can instantly position itself away from obstacles, choose an orientation and initial speed, and resume operation immediately.

(4) The dynamic nature of the robot's run

These desirable characteristics for a mobile robot are all exhibited even in very dynamic real-life situations. When faced with a new set of conditions, the robot does not stop and 'think' but acts instantly to adjust to the changes. The dynamic nature of the robot's actions is especially clear when the robot encounters another mobile robot in action in a confined passage such as the one shown in Figure 2.3. Our repeated experiments show that R2 maintained a roughly straight course while avoiding walls and the bottleneck. It did so despite the complete lack of centralized structure that evaluates situations at hand, plans for the 'best' action sequence to take, and monitors the execution of the chosen set of actions. The nature of the exhibited intelligence here is in line with the notion of intelligence as an observed side-effect of a dynamic process^{6,7}. If properly organized, an agent structure embedded on-board a fully autonomous robot can dynamically yield the necessary intelligence to cope with events, situations, and the intended or implied goal at hand.

(5) The open nature of design

The program that runs the R2 robot is not based on a fixed model but on a form of a volatile dynamic model that emerges from time to time as the robot tries to adjust its behavior to its changing relationship with the environment. In other words, the model is, as is the robot's behavior, emergent as a result of self organization. Since it is not fixed, the robot can perform a wider range of adaptive behaviors, most of which is not even

designed nor intended by the designer of the robot or that of the agents. Even with a relatively small number of agents, the full combination of all internal states of each of the agents would rapidly grow with the number of agents. And because of the non-linear relationship between input and output parameters of such agents, the resulting dynamism will have a very wide range in a number of dimensions. That non-Cartesian robots have dynamism and adaptability beyond those perceived by the designer of the robot gives rise to the assumption that the robot is capable of adapting to more open situations than generally expected. In fact, as described below, the R2 robot has demonstrated its ability to cope in unexpected situations beyond what was originally intended for the experiment.

The R2 robot was eventually pitted against a T-2 robot in the same narrow passageway, as shown in Figure 2.4. The T-2 robot, which has similar dimensions and motion characteristics, was equipped with a similar set of sensors and agents as the R2. Now, not only do both R2 and T-2 have to negotiate a narrow passage, they have to deal with an additional moving obstacle which, if placed near the center of the path, is singularly capable of blocking the progress of the other robot. Without additional agents, the two robots were capable of avoiding and passing each other despite the fact that the relative speed between the robot and the obstacle is now about double. Speaking of the speed, the total time required to execute the code shown in Figure 2.2 is less than a millisecond, even if all the agents got activated in serial in this simulated parallelism set up. Since the active infrared (IR) sensors return the strength of reflection from obstacles at 30 Hz or every 33.3 millisecond, theoretically the robots could pass each other at more than 30 times the current speed of about 30 cm per second per robot, should one find strong enough motors to drive the robots and a way to construct the robots so that they can withstand very sharp turns.

The point to be made here is the achievement of the unprecedented code efficiency of the program and the program's even more remarkable performance when carrying out tasks which are known to be very difficult or impossible in conventional programming. When traveling the wider section of the corridor, typically a quick 'negotiation' takes place between the two robots and a set of two complementary spaces between the opposing robot and a wall is secured. Through repeated and quick adjustments of the heading and the relative position, each robot can find sufficient space and pass each other uneventfully. The result is impressive and amazing in a way, as the two robots repeatedly pass each other smoothly as if they are living animals with ample intelligence to negotiate the passage. When the agents were designed, there was not even a plan to try the robot against a moving obstacle. Yet, the same set of agents for solo runs collectively manage to yield enough intelligence to cope with this greatly more dynamic situation.

When the two robots meet at the bottleneck, as illustrated in Figure 2.5, each robot attempts to push its own way, and a deadlock between the two robots ensues. By adding a simple agent to R2 (called *back-off*) which forces the robot to back off for a few

seconds when it frontally faces a moving obstacle, the R2 robot now momentarily retracts upon a face to face meeting with the T-2 robot. Because space is offered in front of the advance-only T-2 robot, it now proceeds. Depending on the situation, the R2 backs off further, often pushed backward past the bottleneck, and eventually leaves enough space on its side for T-2 to pass by. After the T-2 moves past R2, R2 can now proceed past the bottleneck.

(6) The graceful degradation

The R2 robot with the program shown in Figures 2.1 and 2.2 also demonstrated impressive graceful degradation characteristics. If the *veer* agent stops working for any reason (e.g., failure or deterioration of sensors, wrong threshold, electrical connections), the robot is still capable of making somewhat more jagged turns to avoid obstacles using the *turn* agent. If the *turn* agent and not the *veer* agent loses its ability, the *veer* agent would try to handle all avoidance turns except the ones due to collision with an obstacle, in which case the *bump* agent would generate recovery turns. The robot in this situation would travel among obstacles trying to avoid them by making only gradual turns that the *veer* agent generates. However, such large turns would sooner or later result in a collision with an obstacle since some of the obstacles are bound to be near the robot's course and the *veer* agent would not be able to issue sharp enough turns to avoid them. If the *bump* agent survives while both *veer* and *turn* are dead, the robot would move *forward* until it hits an obstacle, and then try to continue moving forward after the *bump* agent forces it to turn away from the obstacle. This will result in a ricochet trajectory as often seen in behaviors of partially incapacitated insects or those of animals in stress. If only the *bump* agent survives, the robot would still try to avoid touching by a human by swinging the body away from the touch, like an animal seriously wounded which still tries to avoid human contact.

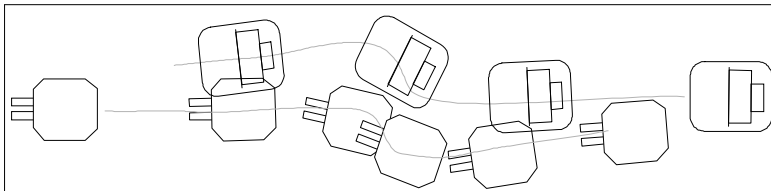


Figure 2.4 Two robots passing each other in a narrow corridor

There are more possible combinations of able and disabled agents and in each case the robot will exhibit partially compromised behaviors that reflect the loss of specific

agents. This is unlike most conventional computer-driven systems which, in most cases, stop working altogether upon an encounter with even the slightest of faults. In non-Cartesian programs, the system would continue to yield some actions towards the achievement of the effect of the original set of actions. Like animals, when deprived of some of their faculties, the system would still continue to maintain behaviors as close to the complete behavior set as possible. This form, level, and extent of graceful degradation has not been achieved in conventional computerized systems. But as shown here, it can be realized rather easily in non-Cartesian programming.

(7) The collection of agents as an autonomous system

The R2 or T-3 robot with agent structure shown in Figures 2.1 and 2.2 is a fully

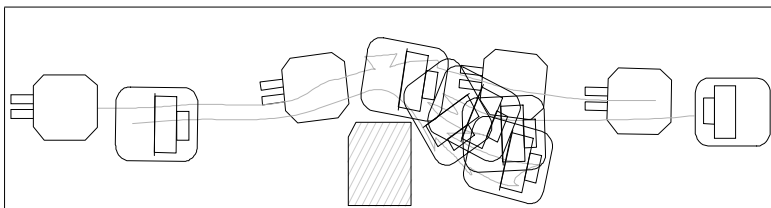


Figure 2.5 Two robots passing each other near a bottleneck

autonomous system in that there is no mechanism that exerts external control over its operation. Stimuli received through sensors can invoke one or more of the agents, which in turn moves the robot to a position itself where more stimuli might become detectible. If there is no change in stimuli pattern and/or level, it will stay put until a change in situation is detected. The system continues to operate autonomously just by reacting to situations at hand. The <if ... then ...> rules (coded in the example of Figure 2.2 as <Whenever (*condition*) (*action*)>) resemble in their appearance a production rule used in *expert systems* or *knowledge-based systems* of two decades ago. However, there is a crucial difference between the conventional AI systems mentioned and the system I have presented here. In fact, the driving principles of the two approaches to programming differ very significantly. In knowledge-based/expert systems, so-called 'reasoning' is said to take place as the result of the invocation of rules by a central execution control facility often called *inference engine*. It typically *forward chains* or *backward chains* production rules, and inference is assumed to occur when each of the chained production rules is explicitly executed.

In the execution of a non-Cartesian program, there is no involvement of a centralized facility such as an inference engine. There is no main routine or task

scheduler to govern the execution of the agents. Each production rule can be invoked only by an event external to it as it is detected in the condition clause of the production rule. The condition clause often contains a description of the relationship between two or more sensor values, and/or sensor value(s) and fixed threshold(s). Values obtained through sensors assigned to the rule during the time of the rule's definition thus dictate the way the rule gets invoked.

The example non-Cartesian program discussed above demonstrates much of the preferable characteristics of the approach. It is clearly different from conventional programming in a number of ways. In seeking resemblance to conventional programs, one notices that each of the agents shown in Figure 2.2 runs like a device handler in a real-time operating system. Both an agent in non-Cartesian programming and a device handler in conventional programming are independent from each other and each has a priority assigned to it.

2.4 Nature of non-Cartesian programs

The manner in which the R2 and T-2 robots are run is quite unique in the field of intelligent mobile robotics. In conventional mobile robotics, robots are the target of *control* and their motions are under tight management of a control program built for that purpose. Robots run with non-Cartesian programs are an autonomous system. The fundamental difference manifests in almost all aspects of the running of the two types of robots. It is the algorithm that generates intelligence in conventional programs, whereas in non-Cartesian programs intelligence is only a side-effect of a process that self-organizes a series of dynamic equilibria in response to changing relationships between a system and its environment. Table 2.2 compares these and other aspects of the two approaches in robotics.

Table 2.2 Comparison of Cartesian and Non-Cartesian intelligent robots

Aspects	Cartesian intelligent robot	Non-Cartesian intelligent robot
Nature of operation	<i>Control</i>	<i>Autonomy</i>

Identification of entities involved	Definitions - Defining robot's position, work, operational environment movements, actions, and tasks	No definitions - Position, motion, plan, effective spheres of operation, and even goals are <i>emergent</i>
Method of representation of entities in robot	Models to visualize, structure, and document the defined entities	No models - instead, a robot maintains a collection of agents to cope with reality
Bases of generation of robot's actions	Measurements - position, velocity, momentum, mass, force, all need to be measured frequently and continuously	No measurements. Humans as an intelligent agent, for example, do not measure parameters in their execution of actions but seeks affordance
Nature of processing	Computation to verify compliance of robot's actions to the definitions and models through measurements	No computation but reactive invocation of agents to respond to changes in operational environment
Method of generation of action sequence	Planning - tasks and a detailed sequence of actions in each task need to be planned in advance mostly in explicit terms in accordance with preset goals . Any deviations from a plan must be checked and corrected through frequent measurements .	Emergent goals and plans. Serious real world tasks are too complex to be explicitly planned. Only rough task sequences (an agenda) could be maintained. The actual sequence of actions <i>emerges</i> through self-organization involving robot's condition and operational environment
Method of obtaining intelligence	By executing algorithm prepared to underwrite a plan	No algorithm -intelligence <i>emerges</i> as a result of self-organization involving <i>agent-invoked</i> processes

Format of execution	Explicit and/or implicit centralized control of execution of tasks and modules	No centralization nor control - spontaneous, asynchronous, and parallel invocation of independent <i>agents</i> in software or hardware
Distribution of action generation mechanism	Overt or covert singularity in system	No singularity - agents that implement processes can be distributed as needed through a system
Method of managing levels of abstraction	Explicit functional hierarchy designed and implemented in control structure	No predefined hierarchy - hierarchical structures may <i>emerge</i> dynamically in operation but none permanent
Nature of system	<i>Closed</i>	<i>Open</i>

3 Implementation options of non-Cartesian programming

I suspect that there are more than a few cases of earlier attempts of non-Cartesian programming prior to Brooks' Subsumption Architecture. However, it was Subsumption Architecture that shed a clear light on the unique characteristics and capabilities of this type of programming, and made researchers and practitioners aware of the potential of the approach. That he implemented the concept as programs to run intelligent robots made it easy to manifest the true nature of the approach. Since his early attempts using Augmented Finite State Machines (AFSM), a large number of programming techniques in this approach have been tried successfully in running a wide range of intelligent robots, demonstrating the versatility of the approach and the coherence of the principles behind it. Below is a description of some of the options successfully attempted and still in use in implementing a 'program' that realizes a non-Cartesian process structure.

(1) Behavior Language option

In 1990, Brooks developed Behavior Language (BL) as a superstructure to LISP¹¹. It has a syntax similar to LISP but with an added mechanism to define *behaviors*, or a causal relationship that links a condition described in terms of sensory input(s) to actuator output(s). Programs written in BL translate into LISP, then into assembler of a target computer. After assembly, the object module is linked to a customized real time operating

system and supporting library modules prepared for the program, and then downloaded onto a microprocessor or micro-controller onboard a robot. The robot then executes the *behaviors* as defined under the supervision of the customized operating system within a simulated parallelism framework.

The BL system has been used in university courses, workshops, and engineering practices in industry for several years. It gave a convenient ready-made mechanism for implementing Subsumption Architecture, the dominant non-Cartesian programming method. Despite its shortcomings (basically integer arithmetic with limited ability to express equations, small number of available target processors despite the language system's multi-target architecture, quasi-multitasking architecture that depends on polling), it has been accepted as a convenient tool to create small-scale software structures that get embedded onto a behavior-based intelligent system with remarkable results. In our research alone, BL has been used in^{9,12-16}. The language is still in use today in the development of software for some smaller mobile robots.

(2) LISP option

LISP is naturally non-Cartesian. Functions in LISP are all written as a causal relationship (as are functions in many other languages, but here in a very visible and consistent manner). LISP enforces a syntax that emphasizes the compactness of a causal relationship as the unit of programming, and this is also the unit construct in non-Cartesian programming that describes an agent. For this reason, a few attempts were made to build a software structure for behavior-based robots. In 1991, Koza implemented a notion of Subsumption Architecture in LISP on intelligent robots in simulation to demonstrate the power of Genetic Programming in evolving robots' behavior^{17,18}.

(3) L option

L is a dialect of full LISP language developed by Brooks in 1993¹⁹. It is a language system which allows incremental compilation. It is considered as a successor to BL with additional features. Automatic generation of real time operating systems is also a main feature of the system. The processor-language structure was implemented typically using a Motorola 68332 processing board. The combined structure has been used in a number of behavior-based robots such as Hermes²⁰ and Pioneer²¹ and in some non-robotic realtime intelligent system applications.

(4) C option

Most high-level symbolic languages, such as C, can be used to build an agent structure

for a non-Cartesian program. In fact, almost all programs written for intelligent robots built by us have been written in C for the past few years. Functions (subroutines, procedures, or functions in other symbolic languages) are defined to express a causal relationship typically using an *if* statement. A *main* routine that governs the syntactic structure of the program is required in almost all high-level languages for a program definition to be complete. A program can only then proceed to *linking* the library and other logistic modules that compose an infrastructure for its run-time environment. Under this strictly von-Neumann top-down provision, some manipulation of the modules will be necessary to make a program non-Cartesian. For example, the *main* routine may have to act as a mechanism to support the *simulated parallelism* operation by monitoring invocation criteria of all agents in the system and invoking ones which meet the criterion. Or it could simply poll all defined agents serially and let each agent decide whether it is to be invoked during the current cycle or not. In a more grandiose manner, each of the agents in a structure can be defined as a task and let a real multi-tasking operating system look after management of the simulated parallelism. In any of these cases, as already shown, agents in non-Cartesian programs are all very short and sufficient effects of parallelism are obtainable out of the respective simulated parallelism attempt. Figure 3.1 is an example of a large-scale, polling-based, non-Cartesian application written in C, with the exception of a few agents written in assembler. The program runs a motorized wheelchair autonomously through a complicated real world environment using landmark navigation. The total size of the software, including the vision processing, is only about 57 KBytes.

(5) Assembler option

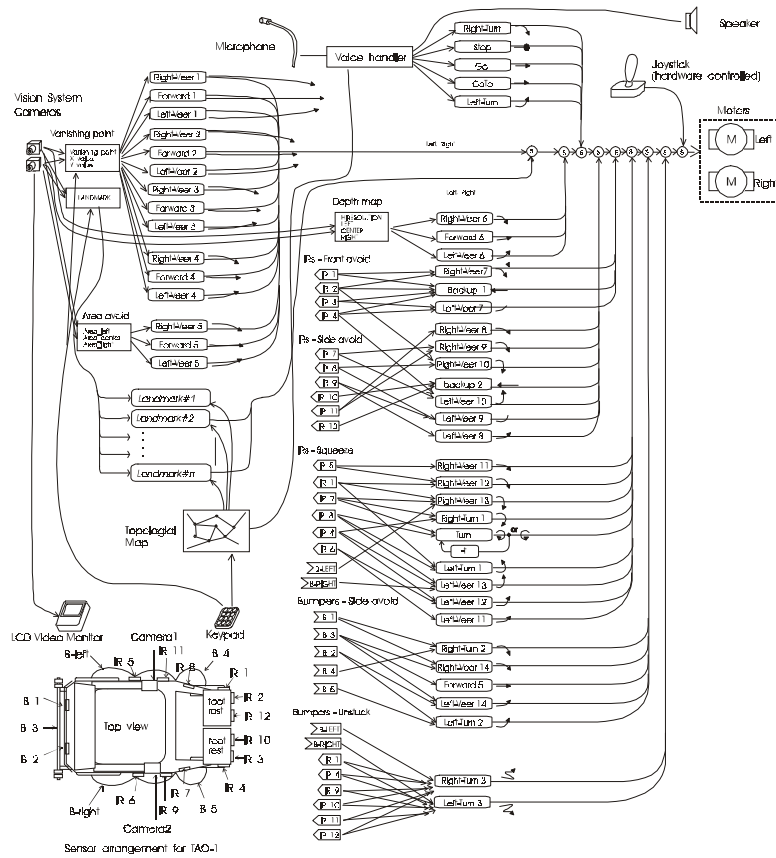


Figure 3.1 An example of a large non-Cartesian program implemented in C

The non-Cartesian process architecture can easily be implemented in assembler languages. The same reasons for using languages which allow symbolic "high level"

representation in conventional programming also exist in non-Cartesian programming. They are: lessening of cognitive load, efficiency in inter- and intra-programmer communication during development and maintenance, and the clarity of programs for explanation to the third parties that aid the management of the program development and maintenance processes. Nevertheless, as in conventional programming, certain programming tasks just have to be written in an assembler either due to unavailability of a suitable symbolic language for implementing minute tasks to be performed under tight temporal or spacial constraints. This was the case in some of our experience when we tried to implement a Subsumption Architecture structure on a vision processing board built around a DSP (Digital Signal Processor)^{13,22}. In these cases, an efficient non-Cartesian program structure was obtained using assembler, with very satisfactory results.

(6) Neural network option

Several neural implementations of behaviors on robots have been attempted²³⁻²⁶. They are non-Cartesian in that they satisfy most of the characteristics shown in Table 2.2 above. Certainly, they are not Cartesian robots in that they do not render themselves to constant control from a system above such as a human operator or a fixed main routine. In fact, it seems easy and natural to materialize a non-Cartesian process structure using neural networks. However, there is a drawback in this approach in terms of lost agility and responsiveness when compared with a collection of reactive agents as discussed so far or as in Braitenberg vehicle²⁷. Each time input patterns change, a considerable amount of time is needed until output of the network for the input pattern is obtained. It is suspected that the loss in responsiveness is due to processing time within the neural network.

The software implementation of a neural network is typically done by having a learning rule such as back propagation implemented in a language like C. This borders on the 'execution of an algorithm' and results in slowed down processes. The hardware implementation of non-Cartesian programming combined with neural network technology could be achieved in the form of a collection of advanced neural network chips in which their ability to detect learned patterns can be executed in an insignificant period of time. The desirable speed of processing of such neuro-chips is of course relative to the speed of the rest of the non-Cartesian processing, such as the time required to switch context at task level when a detection by a neural network occurs, or overhead caused by other so-called housekeeping functions conducted around the chip assembly. The effect of combining neural networks and non-Cartesian programming has not been fully explored except for a few attempts including our preliminary experiments^{13,28}. This is expected to be a rich area of research to be fully investigated and analyzed in the future.

In some of the experiments we have conducted, one or more neural networks

have been downloaded onto a robot so that it can run host-free²⁸⁻³⁰. The additional module strength and lowered dependence on external support the robot exhibits due to the embedding of software gives a heightened sense of autonomy.

(7) Graphic programming language option

Graphic programming languages are an effective tool to implement non-Cartesian programs. Developed mostly for conventional realtime applications such as process monitoring and control, they are also suited for describing process structures in non-Cartesian programs. An agent or a component behavior can be easily described often merely by clicking on icons displayed on the graphical list of available process types. LabView and PACLIB are good examples of this line of development tools readily available on the market. When using these tools, however, care must be given in composing processes in order to make the system truly non-Cartesian. The urge to introduce a procedural arrangement of icons to achieve the effects perceived in the mind of a programmer is very enticing. It is easy to visualize "what might be happening in the heart of an intelligent system" and think procedurally from there. This prevents one from freely composing a non-Cartesian program. Instead, the system must be broken down into a collection of small independent, self contained task-achieving processes. The icons that represent these processes needs to be structured in such a way that, when activated, they collectively form a cluster of parallel processes with minimal or no direct interactions among themselves but only interactions through their actions exerted onto the environment. When activated they should run in parallel and asynchronously, driven mostly by events which occur in the environment.

(8) The AFSM option

AFSM is a term used as a building block by Brooks when he first introduced Subsumption Architecture^{3,31}. It is a finite state machine (FSM) with extensions such as a mechanism that allows the replacement of an input bit string with an alternative stream, suspension of output by an external signal, and auto-triggering of an event using a built in clock. Although a system of AFSMs can be designed manually, in the late 1980s Brooks used to maintain a behavior compiler which generated block diagrams of AFSMs which output diagrams detailed enough for immediate assembly in hardware. A number of intelligent robots were built this way³¹⁻³³. This approach to programming has not been pursued in recent years possibly because the granularity of the building block is too fine to depict agents as the total agent structure has grown considerably since the earlier days of Subsumption Architecture robots. Figure 3.2 is an example AFSM structure for an intelligent robot³⁴.

(9) The discrete component option

The non-Cartesian processing structure can also be implemented using discrete passive and active electronic components without going through the AFSM formality. Such electronic or electro-mechanical components as transistors, diodes, capacitors and registers, and even micro-switches and electro-mechanical relays are often used to build a non-Cartesian agent. A simple pressure-sensitive device can be embedded, for example, inside a bumper made of urethane and attached to the front of an autonomous vehicle. Upon contact, the sensor inside would generate a pulse, which is sent to an analog-to-digital (AD) circuitry, and after amplification, converted into a set of drive currents sent to a steering motor. A set of discrete electronic components that supports this signal

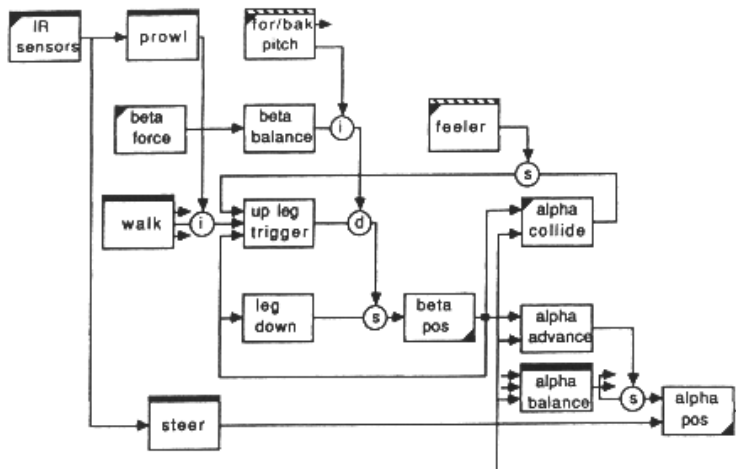


Figure 3.2 Example AFSM structure for an intelligent robot³⁴

conversion process is an agent. A relay, having a primary and secondary circuit can readily represent a causal relationship, thus qualifying as a potential agent in non-Cartesian programming.

(10) The PAL option

As known, Programmable Array Logic (PAL) contains a set of components each of which performs a basic logic operation such as AND and OR. Each of these logical operators can run totally independent of the others. By programming these gates to depict a desired causal relationship, one can easily and very compactly implement a wide range of non-Cartesian programs including one that supports the behaviors of a small

robot. In 1987, Jonathon Connell, then of MIT implemented a set of agents on a PAL³⁵. The PAL is then mounted on a set of two modified toy radio-controlled cars. Four active infrared sensors were added to each car along with a PAL. The set of less than ten behaviors implemented on the device using some two hundred gates included ones that made the car follow a heat source and one that pushed the car back in the presence of an obstacle. The combined behavior mimicked the behavior of a kitten which plays with a human hand. The car would chase a human hand to the point of almost touching it, but the moment the hand got too close, it would quickly retract. As explained next, implementations similar to the successful attempt by Connell are now being reproduced by others elsewhere using similar in principle but more sophisticated hardware for describing logical relationships,.

(11) The FPGA option

In recent years, some researchers began to use Field Programmable Gate Arrays (FPGA) for generating robot motions^{36,37}. FPGA is a collection of a large number of logic gates on a Very Large-Scale Integration (VLSI) chip. Input and output terminals of these gates are under the control of an on-chip processor. The user sends a set of instructions to this processor to have the desired connections established. Because the unit of non-Cartesian programming is a description of a simple causal relationship, it can be implemented by a combination of these logical gates. In applying FPGA to run behavior-based robots, input and output signals need to be formulated and pre-processed to meet the logical signal level and format defined for a specific FPGA device. In a typical robotic application, a set of sensor inputs are linked to the input side of the gate assembly after analog-to-digital conversions and level normalization. For example, a 3.2 volt blip at a bump sensor that means "left bumper touched" would be converted into a "logical on" on one of the input terminals of the FPGA. The condition that "infrared reflection is stronger than a threshold" would be connected as logical on or off on another input terminal. The logical output(s) from the gate assembly is converted into a necessary control signal and amplified before being sent to an actuator. Although the pre- and post-processing could be cumbersome, this way a large number of agents can be implemented on a single FPGA chip with regularity and at a very high density. A more important advantage here is that the logical structures involved will be placed under complete software control.

In conventional Cartesian processing, FPGA is just a very compact way of assembling a large number of logical gates mostly for sensing/control applications. It has been used as such in industry for the past few years and its use is expected to grow due to its convenience as an easily reconfigurable device. Designers of devices with a complex logical structure or simple but a large number of logical gates can now speed up the design-implementation-test cycle by putting a good part of the effort in software. Using the latest FPGA, the reconfiguration can be executed even in run time. The term 'reconfigurable computation' is coined around this device for this reason. However, it is not computation itself that is reconfigured in the conventional sense because it is only a control structure around conventional processors (CPU's) that can be reconfigured using this device.

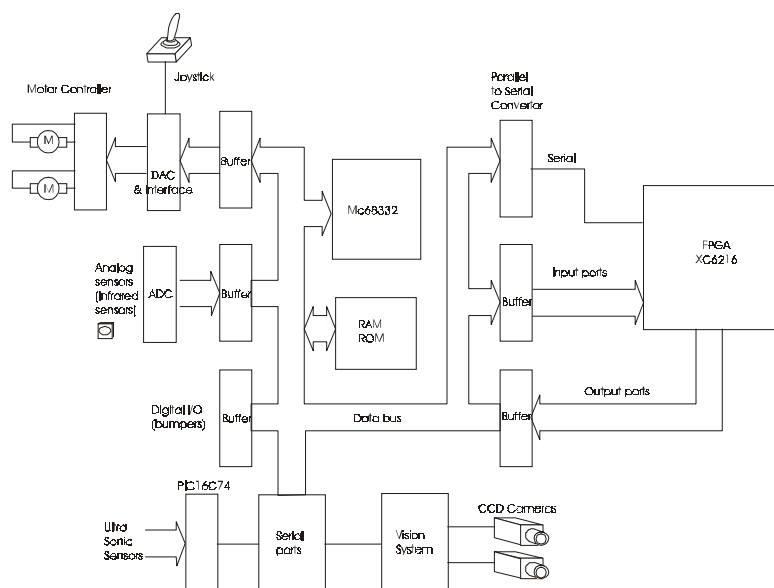


Figure 4.1 A framework for intelligent autonomy management for non-Cartesian programs with evolution

In non-Cartesian computation, the gates configured into agents themselves are the elements directly contributing to 'computation.' Thus the term reconfigurable computation carries a deeper meaning here. The researchers are now exploiting this potential further into a new dimension by applying evolutionary computational methods such as GA and GP to automatically reconfigure the FPGA. The new field of

Evolutionary Robotics (ER) is viewing FPGA as a device for implementing a form of hardware evolution. The scheme is to automatically evolve the autonomy management structure of a robot at the hardware level and to successfully evolve connections between gates, inputs and outputs adjusted during operation. Figure 4.1 is a block diagram of the hardware structure to support this non-Cartesian 'software' structure we have developed and is being applied to a range of embedded systems from a miniature robot for experiments in Evolutionary Robotics to hardware evolution of behaviors on-board intelligent wheelchairs we have previously developed for the handicapped ³⁰.

(12) Which option to choose?

The point to be made here is that in a non-Cartesian processing architecture, it is neither computer hardware nor software, let alone the medium, language or the detailed manner with which they are constructed, that signify the identity of the approach, but the architecture itself. As long as the architecture is maintained, it can be constructed using any of the methods discussed here and more. They can be combined at will to satisfy specific constraints a system must meet, as long as the local interface logistics are satisfied.

5 Benefits of non-Cartesian programs and programming

The example discussed in Section 2.3 and implementational issues covered in Section 4 above demonstrated some of the key aspects of non-Cartesian programs. In this section, I would like to summarize them and also like to describe other not so visible benefits of non-Cartesian programs and programming.

(1) Choices in implementation

It is advantageous to be able to implement the necessary process structure either in hardware or software, or in their mixture. The choice can be casual, such as the availability of suitable components; or practical, so that the implementation fits in available real estate; or theoretical, such as use of FPGA for serious hardware evolution. By having a wider range of selection, one can choose the format that best fits the application. In typical applications of non-Cartesian programming today, implementations that depend solely on software would satisfy most of the cases. However, in cases where space is limited, as in mobile robot applications, an optimal balance between hardware and software, selection of the best processing hardware, and availability of a language system play a significant role.

(2) Very small size of implementation

The programs or hardware structures implemented in non-Cartesian manner are a collection and encapsulation of small element causalities (*agents*) that often take the form of a production rule. As explained above, the <if ... then ...> rules are easily implemented in a large number of readily available computer programming languages and hardware arrangements. The size of each rule is normally very small, often occupying no more than a few to several hundred bytes of memory regardless of the language used. The total size of the program often amounts to no more than a few kilobytes. The size of a program that manages to run an intelligent motorized wheelchair that we developed, complete with vision processing for collision avoidance and landmark-based navigation²² was about 51 kilobytes. The frugality is also the case when the implementation is in hardware. The size of the boards used and the number of components can get as small as one tenth or less of the hardware structure for similar functions implemented in conventional manner.

(3) Incremental nature of program development

The agents are implemented one at a time, either in software or hardware. Whereas a good deal of the entire system is necessary for the system to be tested in conventional system development, after implementing each agent, the entire system can be tested with the new addition. This means that the development of a system can be strictly incremental. An agent could be coded and integrated into the system one at a time without much regard for the history of the development. Contradictions between the existing body of agents and the one just added will manifest the moment the new system is tested, signaling a problem and clearly identifying the source of the problem. These are software engineering benefits one cannot expect in conventional programming where module structure has to be defined and built before coding starts.

(4) Modularity

The modules in non-Cartesian programming are modular in an ideal sense in that they do not have inter-module connections. They deal with each other only through the operational environment. There are no direct linkages between the agents other than those going through the environment. Thus, the program modules can enjoy very high *module decoupling*. It would also have very high *module strength* since a module contains only one fully independent causal relationship and nothing else. In short, this form of programming realizes ultimate modularity - maximum module strength and minimum module coupling. Unlike in object-oriented programming, which is a highly modularized version of Cartesian programming, there is no explicit communication or

exchange of controls between modules.

(5) Evolvable structure

As mentioned in Section 3 (11) above, a new approach to develop agents that generate desired behaviors on robots using evolutionary computation is called Evolutionary Robotics. The field was established less than a decade ago³⁸. Although earlier developments were dependent on software implementation of evolutionary computation methods such as GA, in recent years FPGA's are increasingly used to implement Evolutionary Robotics and other evolvable systems on hardware. Either way, because of the simplicity and the uniformity of the agent architecture that non-Cartesian programming is based on, it is greatly simpler to apply evolutionary computation methods to non-Cartesian programs than to conventional counterparts.

6 Issues with non-Cartesian programs and programming

(1) Non-Cartesian programming is non-algorithmic programming

As is obvious from the example examined above, the method of constructing a computer program discussed here is non-algorithmic. Whatever small steps that describe an agent or a component behavior is too short and simple to be called an algorithm (technically one can, but that is not the essence of the dynamic actions that take place when the module is activated). Programmers and system designers well abreast of conventional procedural programming would likely experience difficulties when grasping the concept, designing a system, and then implementing it. This is only because we have been systematically trained to think in terms of algorithms. The puzzlement a programmer would experience is not unlike the one felt by many programmers when faced with declarative programming such as the one with Prolog language over a decade ago. However, if one looks at the new programming as a process of simply identifying and documenting causal relationships that exist between inputs and outputs, as opposed to seeking procedures necessary for converting inputs to outputs, the confusion is well within the range of containment.

(2) Unruliness of the generated processes

The nature of the dynamic processes that happen when a non-Cartesian program is executed, is unruly. It is a very drastic departure from the way we are used to looking at programs. Programs governed all the actions, effects, and side effects that a computer generates to the last bit and to the finest system clock. In contrast, in non-Cartesian programming we depend solely on the side effects of activated simple agents operating

over non-linear relationships. In a simple case such as the one described in Section 2.3, most, if not all, effects and side-effects of the processes that self-organize as a result of invoking the agent structure are containable. Impacts of a new agent incrementally added are immediately recognizable and its parameters can be easily adjusted according to observations. And one can even direct these effects and side-effects towards a useful goal, as shown in the example. However, in more involved cases, the process of creating a system based on non-Cartesian programming gets more tedious. There will be a large number of cut-and-try cycles of adding a new agent, testing it within the entire agent structure, modifying the agent, and then testing it again. The total amount of effort required to complete a system with similar functionality, however, is still far less than would be required using a conventional approach. Nevertheless, the uncertainty perceived in the method can work against the psyche of system designers who are trained in a certain way of viewing a system and system design. Evolutionary computation is an effective way to contain the processes and automate the process of perpetual and mundane process of revisions.

(3) Difficulty in explaining behaviors

The sequence of minute events that happen in the dynamic process that results from the execution of non-Cartesian programs is often not analyzable nor explainable. The theoretical analysis of the approach has barely begun as a part of the study of *complex systems* and of chaos. The process is viewed instead at a macroscopic level and only the global effects of the execution are analyzed. It is also very difficult to have a plan in the execution of non-Cartesian programs. Again, the inconvenience implied here is mostly because we view processes in a certain way, namely, procedurally. Therefore, to those researchers and practitioners trained in the conventional norm of analyzing all the unexplainable, the new method will appear less rational and even unacceptable. However, the lack of established methodology to explain the details of the operation is not always an issue at all in practicality. One can still construct systems using the approach and even put them into actual use in real world applications.

7 Conclusions

A new approach to programming computers has been described. With this method, computation becomes not only parallel but also non-von-Neumann and non-Cartesian. It has a number of desirable characteristics, especially when applied to robotics, and some drawbacks, when viewed from the conventional standpoint of computer programming. It will be a while before conventional programming practices are affected by this new approach but there is potential for a development that makes the hereto unmistakably clear concept of programming at least obscured.

Acknowledgments

I would like to thank more than a few dozen programmers, hackers, and hardware geniuses who have worked at AAI as an engineer, a coop student or a trainee, and those who are still there. Without their contribution in implementing ideas, it would never have been possible to visualize the emergence of the new way of running computers discussed here.

References

1. E. Yourdon *How to Manage Structured Programming* (YOURDON inc., 1976).
2. Zelkowitz *et al.*, *Principles of Software Engineering and Design* (Prentice-Hall Software Series, 1979).
3. R.A. Brooks, *A Robust Layered Control System for a Mobile Robot*, IEEE Journal of Robotics and Automation, **RA-2**, 14-23 (1986).
4. S. Forrest, *Emergent Computation*, (MIT Press, Cambridge, MA, 1991).
5. C.G. Langton, *Life at the Edge of Chaos* in Artificial Life II, ed. C.G. Langton, C. Taylor, J.D. Farmer, S. Rasmussen, (Santa Fe Institute Studies in the Sciences of Complexity, Volume X, Addison-Wesley, 1992).
6. R.A. Brooks, *Intelligence Without Reason*, Proceedings of International Joint Conference on Artificial Intelligence (IJCAI'91), 569-595, (1991).
7. R. Pfeifer and C. Scheier, *Introduction to "New Artificial Intelligence"* (Institut für Informatik der Universität Zürich, 1995).
8. R. Descartes, *Discourse on Method*, (Paris, 1637).
9. T. Gomi and J-C. Laurence, *Behavior-based AI Techniques for Vehicle Control* in Vehicle Navigation & Information Systems Conference (VNIS'93) (Ottawa, Canada, October 1993).
10. C.G. Langton, *Preface, The Artificial Life Workshop*, in Artificial Life ed. Christopher G. Langton, (Addison-Wesley Publishing Company, 1989).
11. R.A. Brooks, *The Behavior Language; User's Guide* MIT AI Memo **1227**, (April 1990).
12. T. Gomi *et al.*, *Elements of Artificial Emotion*, Presented at Robot Human Communication (RO-MAN'95), (Tokyo, Japan, July 1995).
13. T. Gomi *et al.*, *Vision-based Navigation for an Office Messenger Robot*, Chapter in Intelligent Robots and Systems, ed. V Graefe, (Elsevier Science, 1995).
14. T.Gomi *et al.*, *The Development of a Fully Autonomous Ground Vehicle (FAGV)*, in proceedings, Intelligent Vehicles Symposium'94 (IV'94), (Paris, France, October 1994).

15. T. Gomi and K. Ide, *Emulation of Emotion Using Vision With Learning*, in proceedings, Robot and Human Communication (RO-MAN'94) (Nagoya, Japan, July 1994).
16. T. Gomi, and J. Ulvr, *Artificial Emotions as Emergent Phenomena*, Proceedings of Robot and Human Communication (RO-MAN'93), (Tokyo, Japan, November 1993).
17. J.R. Koza, *Evolution and Co-Evolution of Computer Programs to Control Independently Acting Agents*, European Conference on Artificial Life (ECAL'91), (Paris, France, December 1991).
18. J.R. Koza, *Genetic Programming: On the programming of computers by means of natural selection*, (A Bradford Book, The MIT Press, Cambridge, MA, 1992.)
19. R.A. Brooks, *L*, (IS Robotics, Sommerville, MA, 1993).
20. IS Robotics, Inc. *Hermes II Software Guide*, (IS Robotics, Somerville, MA, 1996).
21. RWI, *Pioneer User Guide*, (Real World Interface, Jaffrey, NH, 1996).
22. T. Gomi and K. Ide, *The Development of an Intelligent Wheelchair*, In proceedings, Intelligent Vehicles Symposium'96 (IVS'96), (Tokyo, Japan, September 1996).
23. B. Yamauchi and R. Beer, *Integrating Reactive Sequential, and Learning Behavior Using Dynamical Neural Networks*, From Animals to Animats III: Proceedings of the Third International Conference on Simulation of Adaptive Behavior, ed. D. Cliff, P. Husbands, J. Meyer, and S.W. Wilson, (MIT Press-Bradford Books, Cambridge, MA, 1994).
24. D. Floreano and F. Mondada, *Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven Robot*, From Animals to Animats III: Proceedings of the Third International Conference on Simulation of Adaptive Behavior, ed. D. Cliff, P. Husbands, J. Meyer, and S.W. Wilson, (MIT Press-Bradford Books, Cambridge, MA, 1994).
25. D. Floreano and F. Mondada, *Evolution of Plastic Neurocontrollers for Situated Agents*, From Animals to Animats IV: Proceedings of the Fourth International conference on Simulation of Adaptive Behavior, ed. P. Maes, M. Mataric, J-A. Meyer, J. Pollack, H. Roitblat, and S. Wilson, (MIT Press-Bradford Books, Cambridge, MA, 1996).
26. R. Naito *et al.*, *Genetic Evolution of a Logic Circuit Which Controls an Autonomous Mobile Robot*, in proceedings, International Conference on Evolvable Systems: From Biology to Hardware (ICES'96), (Japan, October 1996).
27. V. Braitenberg, *Vehicles, Experiments in Synthetic Psychology* (Cambridge, MA: MIT Press, 1984).

28. K.M. Woon, *Landmark Detection Using Neural Networks*, Proceedings of IEEE Singapore International Symposium on Control Theory and Applications (Singapore, July 1997).
29. T. Gomi, *Fully Autonomous Ground Vehicle (FAGV) for Industrial Applications*, Proceedings of International Conference on Robotics, Vision, and Parallel Processing For Industrial Automation (ROVPIA'96) (Malaysia, November 1996).
30. T. Gomi and A. Griffith, *Developing Intelligent Wheelchairs for the Handicapped*, To appear in Lecture Notes in AI: Assistive Technology and Artificial Intelligence (Springer Verlag, 1998).
31. R.A. Brooks, *A Robot that Walks; Emergent Behaviors from a Carefully Evolved Network*, Neural Computation, Vol 1, No. 2, (Summer 1989).
32. J.H. Connell, *A Behavior-Based Arm Controller*, MIT AI Memo 1025, (June, 1988).
33. R.A. Brooks and A. Flynn, *Robot Beings* IEEE/RSJ International Workshop on Intelligent Robots and Systems (IROS'89) (Tsukuba, Japan, 1989).
34. R.A. Brooks, *A Robot that Walks; Emergent Behaviors from a Carefully Evolved Network*, AI Memo 1091, February 1989)
35. J.H. Connell, *Creature Design with the Subsumption Architecture*, Proceeding of the International Joint Conference on Artificial Intelligence (IJCAI '87) (Milan, Italy, July 1987).
36. A. Thompson, *Evolving Electronic Robot Controllers that Exploit Hardware Resources*, Advances in Artificial Life: Proceedings of the 3rd European Conference on Artificial Life (ECAL'95), Springer-Verlag LNAI 929 (1995).
37. AAI, *Experimental Development in Evolutionary Robotics*, Internal document (1998).
38. Harvey *et al*, *Evolutionary Robotics and SAGA: the case for hill crawling and tournament selection*, in C. Langton, ed., Artificial Life III, Santa Fe Institute Studies in the Sciences of Complexity, Proc. Vol. XVI, (Addison Wesley, 1993).