An Adaptive Control Architecture for Freeform Fabrication

J.C. Boudreaux
Advanced Technology Program
National Institute of Standards and Technology

Introduction

Global competition is driving manufacturers to significantly reduce the length of the product development cycle while maintaining product quality. But this approach has the potential of exposing firms to huge design and engineering revision costs, since even a modest revision may give rise to a cascade of new change requests. It has been demonstrated conclusively, and repeatedly, that a work-faster strategy has the insidious negative effect of forcing workers to search for local optima, which are usually ones in which the burden is shifted either upstream or downstream. A more plausible strategy is to work smarter: by increasing the amount of information available at each stage, and by identifying and correcting problems as promptly as possible. The work-smarter strategy has been the root cause of the phenomenal growth of rapid prototyping technologies and especially solid freeform fabrication (SFF), which includes such technologies as stereolithography (SLA), selective laser sintering (SLS), laminated object manufacturing (LOM), and ballistic particle manufacturing (BPM).

Though differing in details, SFF technologies share a common operational methodology /9/. First, a computer-aided manufacturing (CAD) solid model of the part is constructed. This model defines the part boundary, which separates the interior (material) region from the exterior region. Second, the CAD solid is converted into a model in the STL file format, which is a *de facto* standard originated by 3D Systems. The STL model approximates the part boundary by a set of polygonal (in fact, triangular) facets. This representation permits efficient algorithms for the computation of planar intersections, but the number of polygons needed to approximate curved surfaces grows very rapidly with the geometric complexity of the surface. STL models are also subject to errors caused in part by finite-precision floating-point arithmetic /2/. Third, the STL model is mapped into the machine's coordinate frame and then sliced into vertical cross-sections. The size of the step between successive slices is a scalable parameter which typically varies from 100 to 500 micrometers. If the STL model has been correctly built, each slice will contain one or more polygonal regions. The fabrication process then produces thin polyhedra with the required polygonal profiles either by solidifying liquids or powers or by gluing together solid laminations and cutting along the polygonal edges.

In this working paper, an architecture for the adaptive control of solid freeform fabrication devices will be sketched. Close-loop controllers, such as proportional-integral-derivative (PID) controllers, are already in wide industrial use. Adaptive controllers are based on a feedforward scheme in which the behavior of an inner close-loop controllers is modified, or tuned, in response to predictions based on measured changes in the fabrication environment /1/, /8/. Adaptive control, if successful, would address many thorny issues of freeform fabrication, permitting the identification and use of such process parameters as the traverse speed of the laser, optimal layer thickness, and tolerance bounds on STL models. Generally, the main advantage of adaptive control for SFF is to deepen and extend the predictive validity of fabricated artifacts for downstream manufacturing processes.

The Quality-in-Automation (QIA) Project: An Example

There is a connection between freeform fabrication and more traditional manufacturing technologies. This connection was stated very clearly by Chua Chee Kai in /**9**/, "Rapid prototyping systems are like a new type of numerically controlled (NC) tool. Unlike traditional NC, however, the algorithms needed to generate rapid prototype programs are simple. There are no tool changes to make, no multiple setups, no complex surfaces to follow and no pockets to mill." An interesting observation, but my experience with the NIST Quality-in-Automation (QIA) project suggests that the connection between NC machine tools and freeform fabrication may be closer than Kai suggests. As formulated by Yee and Donmez /**6**/, the QIA system was designed to be a "quality control/assurance system that exploits deterministic manufacturing principles ... based on the premise that most errors in the manufacturing process are repeatable and predictable..." The compensation strategy was achieved by implementing tool-path modification of the machine using a combination of kinematic, geometric, and thermal models of machine-tool errors. These models were obtained by pre-process machine characterization measurements of the machine tools involved in this project, including a Monarch VNC-75 Vertical Machining Center with a GE-2000M CNC controller and a Monarch Metalist Turning Center with a GE-2000T CNC controller. The QIA system supported two operational modes: a real-time mode and a process-intermittent mode.

In real-time mode, the compensation system monitored the machine tool and the metal cutting process, using sensory data obtained from thermal, vibration, force, and ultrasonic sensors, and then using model-based adaptive machining algorithms to do error correction by modifying the machine tool path as well as the speed and feed rates. After the appropriate error correction was calculated, as discussed by Boudreaux in /**6**/ and /**4**/, the required compensations was achieved by sending correction signals to a specialized electronic device, called the Real-Time Error Corrector (RTEC), "which is inserted between the position feedback device for each axis and the machine tool controller. It alters the feedback signal to cause the machine to go to a slightly different position to compensate for the predicted errors" /**6**/, /**10**/.

In process-intermittent mode, the compensation system would identify and measure part errors which were caused by the machining process, but which could not be accounted for by means of the deterministic model. For example, tool deflection or tool wear often cause errors of this kind. These form errors were detected experimentally by on-machine gauging between the semifinishing and finishing cuts. Once detected, the errors were corrected by either by using the RTEC feedback system with on-the-fly (and temporary) modifications of the error model for the finishing cut, or by generating a modified NC part program (understood to be coded in conformity with EIA RS-274-D) for the finishing cut, as discussed by Bandy in /**6**/.

To use part programs in an error compensation scheme some consideration must be given to the methods of verifying and nominalizing them. Verification is the simpler of the two, since it can be accomplished by combining an elementary syntactic and semantic analysis of the part program with the creation of solid geometric model of the finished part and all of its intermediates. This solid model may de generated by joining the volumes created by sweeping the cutting tool profile through the tool path and then subtracting the resulting solid from a solid model of the initial blank. The part program is verified if this solid is indistinguishable from the solid CAD model of the part. Nominalization is more difficult. In fact, there is no general agreement about the conditions which a part program has to satisfy in order to be accepted as nominal. The idea is that a part program is nominal if it would generate an exactly accurate part, satisfying all of the design constraints, if it were executed on a perfectly conformant machine, that is, a machine with no systematic errors. Since interpolation methods are important sources of non-repeatability, it has been claimed that a nominal part program should only contain linear tool paths, and that all non-linear (circular or parabolic) tool paths should be replaced by a suitable linear approximations. It has also been claimed that the segmentation problem requires the off-line programming system to allow access not only to part programs, but also to process plans and CAD data. Such a data-rich framework would allow the programming system to maintain an internal representation of the nominal program so as to correctly propagate changes throughout the segments.

The QIA project suggests the following provisional sketch of an adaptive control architecture for solid freeform fabrication:

Adaptive Control Architecture. *(1) The inner, or core, controller accepts as command input any of an infinite set of syntactically well-formed control programs. Under the action of an algorithmic translator, the controller outputs to the plant a series of actuation signals, thereby causing the plant to perform a corresponding series of motions. In close-loop control systems, the controller has access to feedback signals which record measured deviations between the commanded and actual plant output. (2) The adaptive controller has access to the command input and feedback signals of the inner controller, and also to environmental parameters whose measured values have been correlated with future plant performance. Under the action of an algorithmic translator, the adaptive controller may respond in any combination of three fundamental modes:*

> ***I. feedback adaptation mode,*** *in which the adaptive correction is accomplished by modifying one or more feedback signals (corresponding to the QIA real-time mode);*
> ***II. command adaptation mode,*** *in which the adaptive correction is accomplished by modifying the inner controller's command input while preserving syntactic well-formity (corresponding to the QIA process-intermittent mode);*
> ***III. translator adaptation mode,*** *in which the adaptive correction is accomplished by modifying the inner controller's translation algorithm.*

The third mode poses possibilities which are often discussed in the context of control systems based on neural net, fuzzy logic or genetic algorithm technology /1/, /8/. To make further progress, the notion of programmable devices will need to be considered more carefully.

A Theory of Programmable Devices

Devices are pieces of capital equipment that are used to modify the position, shape, size or material condition of manufactured parts. Examples of devices include numerically controlled machines, programmable fixtures, tool-changing devices, robots, and also the freeform fabrication devices mentioned above. More precisely, a device is an piece of capital equipment which (1) may be resolved into finitely many coupled components, each of which is itself a device, (2) has the capacity to perform a finite, but extensible, collection of industrially significant operations, and (3) may occupy any of a finite number of operational states. The first point says that devices may be nested within other devices, forming structures which resemble connected graphs. Since all devices have characteristic operations, industrial devices must support a concurrent, distributed processing model. That is, every device operation reduces to a series of concurrent operations by its component devices, the relative motions of which are precisely orchestrated. The controller converts these commands into a precisely structured sequence of signals to the actuation system. Thus, any realistic theory of devices must include an identification of the set of elementary or stereotypic motions that the device is able to produce, and also the definition of a method for combining elementary motions into more complicated motions.

A device is programmable if every motion in its behavioral repertory, however complicated, may be produced as a response to a well-formed combination of input commands to the inner controller. There is a close connection between a industrial programmable device and I/O devices such as floppy disk drives or printers. The behavior of disk drives or printers is defined in terms of the programmed commands that then can be given. For example, a disk drive can be commanded to **open** a file or to access a location in a file and return a piece of data, but neither command can be given to a printer, which can only be commanded to perform text or graphics output operations. A programmable device may require non-programmed machine-tending actions on the part of an operator. For example, the part to be machined may need to be properly fixtured or the cutting tools mounted in the turret, but these operations are no different in principle loading paper into the printer or putting a floppy disk in the disk drive.

Command expressions can be represented as a single phrase, consisting of a main verb, the symbolic representation of the operation, and an optional list of one or more modifiers **/5/**. For example, if a suitable coordinate frame has been defined, then one might imagine that a command of the form:

```
(MOVE (TO #(1.0 2.5 -1.0)) )
```

would cause a motion from an initial position to the position specified in the **TO** clause, provided the specified position is in the work volume of the machine tool. Without further qualification, the point-to-point specification of the motion permits an enormous range of acceptable path geometries. This range may be reduced by additional parametrization of the command. If one associates commands with actions by using a template mechanism, then the template mapping algorithm would take all of the parameters into account when generating the native language of the device controller **/3/**. The semantic interpretation of a parametrized verb phrase is the range of behavior which that phrase would cause if it were carried out by any appropriately configured device.

A Computational Framework

The task of any programming language for automated manufacturing is to provide a uniform programming language environment for the construction of control interfaces to industrial manufacturing processes, and an integrated system of software tools for translating product design and process planning specifications into equipment-level control programs. To provide support for the information requirement of programmable devices, an experimental Lisp-based programming language, called AMPLE **/5/**. This system was built in the mid 1980s for use in the NIST Automated Manufacturing Research Facility (AMRF) and was also used to design the QIA Quality Controller described above **/4/**, **/10/**.

AMPLE is an interpreted language. The behavior of the interpreter may be described as an infinite loop: (1) **read** an expression from an input stream, (2) **evaluate** the expression in the context of the symbolic environment (an ordered list of symbol/value pairs), and then (3) **write** the expression resulting from the evaluation to an output stream. The domain of AMPLE expressions is very simple. There are atomic expressions, including characters, strings, integers, floating point numbers, complex numbers, and arrays. When the interpreter is given any of these expressions to evaluate, it returns the expression itself. Symbols are atomic expressions which has a more complicated role. When the interpreter is given a symbol, the evaluation procedure tries to match the symbol with the all of the entries in the *symbolic environment*. If a match is found, then value corresponding to the symbol is returned. If there is no match found, then the interpreter returns a message to that effect. In addition to atoms, the only other class of expressions are lists. When the interpreter is given a list to evaluate, it assumes that the expression is the application of a function, which must be the first component of the list, to the interpreted values of the arguments, which must be the remaining components of the list. The value obtained is then returned as the value of the original list.

The value of an expression depends upon the operation of the interpreter and the current content of the symbolic environment. When initiated, AMPLE comes supplied with a set of primitive functions and variable symbols. This primitive environment may be dynamically altered by subsequent operations of the interpreter. As explained in **/5/**, some functions, when evaluated, cause new symbol/value pairs to added to the environment. For example, the evaluation of an assignment expression can cause the symbol **pi** to be added with the associated value *3.14159*, or the evaluation of a function definition expression can cause a new function to be added. All symbolic environments are persistent objects which can be stored in external files and then loaded back when needed. The normal effect of reloading an environment is to restore the prior configuration, which means that AMPLE programming is focused on the construction of snap-on workspaces. Many workspaces have been developed, three of which are directly relevant to the topic of this paper: concurrency, code generation, and real-time processing.

Concurrency. There are two different issues that an industrial programming system must resolve: what operation, or operation sequence, needs to be performed, and when must the operations be initiated and terminated. The program needs to specify both the operations and the order in which the operations need to be done. Since a single device may actually contain several controllers, and thus different operations can be executed concurrently, an efficient representation of these programs is by means of directed graphs (digraphs).

A set of basic functions for concurrency has been defined by Hoare /7/: **spawn** causes a new process to be created and added to the list of active processes, **block** causes a process to be unavailable for further processing by removing it from the active list, **activate** restores a blocked process to the active list, and **alt**, when given a set of alternatives, causes one of them to be selected. In addition, the function **par** causes agents to bring about explicit parallelism. When **par** is applied to a sequence of processes, called *children*, the process containing the occurrence of **par** being executed, called the *parent*, is immediately blocked. The children are concurrently spawned, and are then processed in the manner just defined. When all of the children have successfully terminated, and only then, the parent is reactivated. In many situations it is necessary for processing agents to coordinate their behavior. This is accomplished by means of *communication channels*. These are special processes which are used as structural members in order to establish a communication link between two or more processing agents, some of which want to participate in an input event, and others in an output event. For example, suppose that there is a Boolean-valued symbol, say **idle**, in the symbolic environment which is to be used to hold the current operating condition of a NC machine. This can be implemented by communication channels. The NC machine, actually a processing agent going proxy for the machine, is allowed to participate in output events with respect to **idle**, and any of a number of agents may participate in input events. The channel enforces a hand-shake discipline, that is, when the machine agent needs to output to **idle**, it acquires the output port, blocks all other input-requesting agents from acquiring the input port. The agent then updates **idle** and releases both the input port and the output port. If input-requesting agents are blocked, either by an update in process or by other inputing agents, they may, at their own option, either queue up to peek at **idle** or simply go about their business. This example shows that communication channels provide an elegant way to implement device states.

Code Generation. A code generator is a program which translates high-level device actions into low-level application programs in the device controller's native language. Code generation, which encourages effective re-use of existing control programs, is typically carried out in three stages. The first stage consist in the elaboration of precise definitions for the target family of devices. These definitions should group devices into classes based on their components, the operations that they can be commanded to perform, and their operating states. The second stage consists of the development of a library of program templates, each of which has been constructed by inserting parameters within carefully chosen fragments of control programs. The third, and last, stage is the definition of combinatorial rules for stringing templates together and uniformly replacing the embedded parameters with appropriate values.

There are many possible template processors techniques. One requires a distinction between a context, which is a linear sequence of characters, and algorithms or chunks of executable code, which are embedded within the context and whose interaction with it can be very complicated. The embedded code is prefixed by a selected special character, such as the **at**-sign @. The template processor scans each character in the source file in the normal left-to-right order and writes the character to the target file. When a special symbol is encountered, the template processor assumes that it is immediately followed by a legal expression, which is to be passed to the interpreter for evaluation. The output to the target file is determined in the course of the evaluation of the embedded expression. In general, all template processors must include a completely unambiguous algorithmic presentation of the *mapping function* which, given any legal template and (possibly) any legal responses from external information sources, will generate the target text in a deterministic manner.

Real-Time Processing. Since all interpretive systems, need to reclaim memory, a process called garbage collection, which is a time consuming procedure, the interpreter has to have an internal *real-time processor* (**rtp**). This processor is a programmable module, acting as a (virtual) coprocessor of the **AMPLE** interpreter. The **rtp** has a limited operation set but a sophisticated set of data structuring capabilities. This processor needs to address two conflicting design goals: it must be able to execute at very nearly compiled language speeds, and it must have direct access to objects dynamically created by the interpreter. This conflict can be resolved by using special off-line programming services to build the objects, which are then accessed and updated using the special access functions within **rtp**. Moreover, to prevent garbage collection, the access and update functions must never cause the allocation of any free memory cells.

Every **rtp** is linked to external consumer systems through **ports**. A port is a finite sequence of bytes which represent the remote communication between these systems together with a protocol which defines input and output behaviors of both systems. A clear example is a communication system in which one system, call it the target, writes instantaneous position information and reads correction signals, and the second system reads the position signal and writes the correction signal.

Any answer returned by **rtp** can be assessed in terms of its mathematical correctness, but real-time computing places an additional burden: the assessment of correctness also depends on a payoff function. For example, *hard real-time* systems are those whose payoff is a non-zero constant value if the correct value is returned within the allowed interval, and 0 everywhere else. In this case, there is a constant payoff for an answer delivered within the interval, but once the time interval has past, the answer has no value at all. A different payoff function might support different strategies. For example, given a payoff function with a non-zero initial value which slowly approaches 0, then there would always be at least a small advantage to returning an answer.

Discussion

The control problem for solid freeform fabrication devices has two classical aspects: the identification of basic operations, and the definition of a methodology for modifying, or adapting, these processes to specific situations. It has been proposed in this paper that the computationally difficult part of this problem can be handled by an very-high-level interpretive adaptive control architecture. This adaptive control family behaves as a system with a modal switch so that adaptation may be accomplished by modifications of the feedback system or the command system, both of which have been extensively investigated, or by structural modifications of the translation system of the inner controller, which is the most difficult case.

Until now, systems with this capacity have been based upon either neural nets, fuzzy logic, or the genetic algorithm. For example, a neural net consists of a network of active elements, called neurons. Each neuron may have several input connections, but all have only one output. Each input connection has an assigned weight. The value returned by each neuron is determined by multiplying each input value by its associated weight, summing these products together, and then by applying a sigmoid function to the computed sum. In this case, adaptation is a matter of updating (that is, incrementally revising) the associated weights. The adaptive control architecture sketched in this working draft suggests an approach to this problem through the design and implementation of what might be called an adaptation-instrumented command translator.

References

1. Astrom, K.J. and Wittenmark, B. *Adaptive Control*, Addison-Wesley, 1989.
2. Bohn, J.H. "Removing Zero-Volume Parts from CAD Models for Layered Manufacturing," *IEEE Computer Graphics and Applications*, November 1995, 27-34.
3. Boudreaux, J.C. "Code generation techniques to support device abstraction," *International Conference on Manufacturing Systems and Standardization, ICMSS '91*, Budapest, Hungary, 1991, 1-8.
4. Boudreaux, J.C. "A Programming Language Environment for Quality Control," *20'th International Programmable Controllers Conference, IPC'91*, 1991, 579-590.
5. Boudreaux, J.C. "Concurrency, device abstraction, and real-time processing in AMPLE," in W.A. Gruver and J.C. Boudreaux (eds.), *Intelligent Manufacturing: Programming Environments for CIM*, Springer-Verlag, 1993; 31-91.
6. Donmez, M.A. (ed.) *Progress Report of the Quality in Automation Project for FY 90, NISTIR 4536*, March 1991.
7. Hoare, C.A.R. *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ, 1985.
8. Ioannou, P.A. and Sun, J. *Robust Adaptive Control*, Prentice-Hall, 1996.
9. Kai, Chua Chee "Three-dimensional rapid prototyping technologies and key development areas," *Computing and Control Engineering Journal*, August 1994; 200-206.
10. Yee, K.W., Bandy, H.T., Boudreaux, J.C., and Wilkin, N.D. "Automated Compensation of Part Errors Determined by In-Process Gauging," *NISTIR 4854*, 1992.