Toward a Computational Theory of Discrete Manufacturing
(Extended Abstract)

J.C. Boudreaux
Advanced Technology Program
National Institute of Standards and Technology
U.S. Department of Commerce

Abstract

The main thesis of this paper is that industrial plants should be modeled as artificial organisms. Like their biological counterparts, plants must be designed to adapt to, and be productive over, a wide range of external disturbances (such as unanticipated trends in the production schedule) and internal error signals (such as performance degradation due to wear). The approach to be taken is the development of a computational theory of manufacturing systems within a formal framework defined by an interpretive model of computation.

Introduction

A characteristic of industrial plants is that the behavior of both the workers and the capital equipment is choreographed to be efficiently responsive to the demands of the external world. The organizational rules which coordinate the behavior of a plant constitute its **manufacturing system**. From a high-level perspective, plants are represented as black boxes which contain technology-dependent production functions to convert input resources into output products that customers want at prices they are willing to pay.

The manufacturing system of a plant should be tuned to a specific mix of end products. The plant receives orders for some quantities of end products to be delivered by a specified date. These individual orders are periodically aggregated into master production schedules. A master production schedule is **feasible** if the plant, operating in a nominal, as-designed manner, can output all of the ordered end products by the scheduled delivery date; otherwise, the schedule is **infeasible**. Thus, a plant is properly tuned just in case the range of master production schedules which are likely to be encountered are feasible. Should the aggregated customer demand begin to drift beyond this range, then the plant's performance is likely to become increasingly degraded and inefficient. Over a longer time horizon, the product mix itself will change: current products will be freshened or possibly discontinued, and new products will be designed and added to the mix. Hence, the manufacturing system that the plant is founded upon is shaped by the need to follow, and ultimately to adapt to, this evolving product mix.

More concretely, the behavior of a plant is dependent upon the operational capacities of the industrial devices that it contains. Industrial devices are items of capital equipment which modify the position, shape, size or material condition of manufactured parts. Examples of devices include numerically controlled machines, fixtures, cutting tool changers, robots, and material handling systems.

Industrial devices have three operational phases. First, the **setup** phase is intended to bring about some specified initial conditions. This phase includes all of the steps needed to ready the device for the operations to be performed, including making ready the workpieces and loading all of the cutting tools. If the device is programmable, then this phase includes the downloading of one or more programs to the device controller. The configuration table of the device may need to be updated; for example, the contents of the tool register would be updated by adding that the 3/4 in ball-nose end milling tool A12767 is now in a specific slot of the tool holder. Second, the **run phase** performs the manufacturing operations on the workpieces. If the device is programmable, then the control programs are queued for execution by the controller. Third, the **teardown** phase either returns the device to some agreed upon neutral state, or does something more complicated such as anticipating the next setup. Of course, the three-phase model is an idealized representation of device operation in a manufacturing system. For example, this model assumes that the devices continue to operate nominally, when in practice this assumption is rarely satisfied. Industrial devices have an inherent tendency to deviate from nominal: surfaces wear against one another, cutting tools break or become dull with use, fixturing clamps loosen, bearings wear out, and so on.

The main thesis of this paper is that industrial plants should be modeled as artificial organisms. Like their biological counterparts, industrial plants have bounding surfaces which separate an interior region from an exterior region. Input resources pass from the exterior region to the interior region through the bounding surface, and, after a suitable lag, end products pass in the opposite direction. The overall behavior of the plant is defined by the interactive behavior of the interior

components. Industrial plants must also be designed to adapt to, and be productive over, a wide range of external disturbances (such as unanticipated trends in the production schedule) and internal error signals (such as performance degradation due to wear).

This paper will develop an initial sketch of a theory of manufacturing systems. The "artificial organism" model implies that this theory will be successful if it generates formal representations of industrial plants whose structural features are emergent properties of the systems of interactions between the plant's components. This is a difficult undertaking since the components have a strong claim to be regarded as artificial organisms in its own right. One might try to code a theory of this kind within a set-theoretical framework, but this approach yields a theory which is very difficult to use. The approach to be taken here is the development of a computational theory of manufacturing systems within a formal framework defined by an interpretive model of computation ([1],[4]).

Computational Domain Theory

A **domain** is a (possibly infinite) collection of entities such that: (1) there is a collection of **atoms**, that is, entities which are stipulated at the outset and which may not be decomposed into simpler elements of the domain; and (2) there is a finite set of **composition methods** which allow compound entities to be constructed from simpler ones. The composition methods are based on a set of operators with respect to which the domain is closed, that is, if **F** is an N-adic operator, then the application of **F** to any sequence of N domain elements is an element of the domain. The distinction between atoms and compound elements is domain-specific i the sense that what has been taken as an atom in one domain can be taken a compound element in another domain (which has its own set of atoms and composition methods).

For example, the positive integers are a domain whose single atom is the integer *1* and whose composition method is based on the addition operator:

1 is positive_int,
$+(x,1)$ is positive_int, if x is positive_int

The set of positive integers can be constructed as follows:

$$1,\ +(1,1),\ +(+(1,1),1),\ \ldots$$

which are the canonical positive integers. Domains also admit constitutive relations. For example, the domain of positive integers admits various ordering relations, including the **equality** relation which holds between terms which may be reduced to the same canonical positive integer, that is:

$$+(+(1,1),+(1,1)) = +(+(1,+(1,1)),1).$$

The addition of positive integers can be handled by repeated applications of the equational rules for commutativity and associativity of addition:

Comm. $+(x,y) = +(y,x)$
Assoc. $+(+(x,y),z) = +(x,+(y,z))$

Thus, $+(+(1,1),+(1,1))$ may be reduced to $+(+(+(1,1),1),1)$ by one application of the associative law. Algebraically, the domain of positive integers is an instance of a commutative semigroup.

Manufacturing domains are more complicated than the domain of integers. Since neither plants nor plant components are static entities, the most plausible approach is to construct a domain theory for them within a framework defined by a Lisp-based interpretive model of computation, which has long been the workhorse of the artificial intelligence community [1],[4].

Lisp interpreters repeat the following sequence of steps: (1) **read** an expression from an input stream; (2) **evaluate** the expression in the context of the current **symbolic environment**, which is a list of symbol/value pairs; and then (3) **write** the expression resulting from the evaluation to an output stream. The atoms of the Lisp domain include characters, strings, integers, and floating point numbers. When the interpreter reads Lisp atoms of this kind, the value it returns is the atom itself. The set of Lisp atoms also includes a potentially infinite set of **symbols**. When a symbol is read, the evaluator tries to match the symbol with an entry in the symbolic environment. If a match is found, then the corresponding value is returned; otherwise an error message is returned.

The compound elements in the Lisp domain are finite sequences of expressions, called **lists**. The list construction operator is the **cons** function: if x and y are domain elements, then **cons**(x,y) is the list whose first element, or **head**, is x, and whose remaining elements, or **tail**, are y. The only list not constructed in this manner is the **nil** list which has no elements. Two constitutive operators provide access to the elements of lists: the operator **car** returns the head element of every list to which it is applied, and the operator **cdr** returns the tail element. The following equations illustrate one important formal connection between these constitutive operators:

$$\mathbf{car}(\mathbf{cons}(x,y)) = x$$

cdr(**cons**(x,y)) $= y$

Lists are used to code not only the compound data structures but also the programs that operate on those structures. If the interpreter reads a list, say x, then it is assumed that x codes a program, specifically, that **car**(x) is a function which is to be applied to the interpreted values of the elements of **cdr**(x), which are assumed to the arguments of the function.

Computational domains are constructed with the Lisp framework by building all of the necessary structural elements into the symbolic environment. Thus, the constructed domain is spread over a list of symbol/value pairs which are dynamically updated by the evaluation of special LISP functions, including **setq** and **defun**, which cause these updates to take place. For example, the following expression, written in the usual Lisp style:

(**setq** *A* (**cons** *1* **nil**))

causes the symbol/value pair *(A (1))* to be added to the symbolic environment and any subsequent evaluation of the symbol *A* will cause the one-element list *(1)* to be returned. Having identified by this means a set of domain elements, the final task is to use the special function **defun**, or one of its Lisp companions, to define the constitutive operators of the domain.

The structural elements of the manufacturing domain are directed graphs (see **[3]** or **[4]** for details). Directed graphs, or digraphs, consists of two sets: a finite non-empty set of vertices, also called nodes, and an ordered set of vertex pairs, called arcs or edges. Arcs always have a direction, that is, arcs always go from one vertex, called the out-vertex, to another, called the in-vertex. Vertices connected by an arc are said to be adjacent. The number of arcs into a vertex (possibly zero) is the in-degree of the vertex, and the number of arcs from a vertex (possibly zero) is the out-degree of the vertex. A path from one vertex to another is a sequence of vertices such that each is adjacent to its immediate successor in the sequence. Digraphs may be implemented in the Lisp domain as lists of vertex cells. The vertices are represented by positive integers called index numbers and the vertex cell is a list of the vertex index, a list of all vertex indices which are adjacent into that vertex, and a list of all vertex indices which are adjacent from that vertex. For example, *(6 (3 5) (7))* is a vertex cell for vertex *6* which has in-arcs from vertices *3* and *5*, and out-arcs to vertex *7*.

The constitutive operators of digraph domains include operators which add a vertex, delete a vertex, add an arc, and delete an arc **[4]**. These operators may be applied dynamically, that is, during process of evaluating a digraph object, the shape of a digraph can be modified by the addition and deletion of vertices and arcs.

Digraphs are combinatorial objects, but the point of introducing this type is that digraphs can be used to provide places to store useful values. The storage method consists of mapping values onto vertices. These values are collectively called the **contents** of the vertices. There are two constitutive operators which allow us to manipulate the contents of a specific vertex. The first operator gets the contents from a specific vertex, and the second puts new contents on the vertex, overwriting whatever was stored there. No general assumptions need to be made concerning the kind of entities which are admissible as the contents of vertices, except that they be evaluatable. The contents could include expressions which, when evaluated, would modify the structure of the digraph and also have side-effects on the symbolic environment in respect to which subsequent evaluation of the digraph is taking place.

Product Domains

Every manufacturing system can be resolved into a collection of **objects**. Objects may be defined as physical entities which are transported from one plant location to another in the manufacturing process. Thus, cutting tools, jigs, fixtures, gages, and other tool-like objects are properly classified as objects. But machine tools and other permanently located items of capital equipment are not.

One set of objects is the set which are destined to be components of end products, for example, stock material items, in-process workpieces, finished workpieces, in-process assemblies, and the end products themselves.

> **Df.** A **product domain** is a set of digraphs whose vertices represent the components of an end product. An arc from one vertex to another indicates that the component represented by the first is required in the manufacture of the component represented by the second, or equivalently that the manufacture of the second depends upon on the prior availability of the first. (1) Plant inputs are represented by vertices which have a zero in-degree. Plant outputs are represented by vertices with zero out-degree. Intermediate components are all components which are neither plant inputs nor plant outputs. (2) The contents of every component consist of a list of attribute/value pairs, including a component identifier, component geometry and other design data, and a process plan which specifies

the manufacturing operations which are to be applied the input components to produce the output component. The input components of the process plan are those represented by the vertices which are adjacent to the component. (4) Every component identifier has one and only one occurrence in a product digraph. Each component has one and only one source vertex.

Without additional stipulations this definition permits the construction of product domains with odd, even pathological, objects.

The first stipulation has to do with an important structural property of product digraphs. Suppose that X is a vertex in a product digraph. Then the vertex Y **supports** X if Y is a adjacent to X, or there is a vertex Z such that Z supports X and Y is adjacent to Z. Since it is intuitively clear that no vertex can be self-supporting, it should be stipulated that product digraphs may contain no paths which connect a vertex with itself, that is, all reasonable objects in product domains must be **acyclic**. This rules out two pathological cases: the case in which a vertex is self-connected by a loop arc which make the vertex adjacent to itself, and the case in which there are two vertices, say X and Y, such that there is an arc from X to Y and an arc from Y to X. These cases are pathological in the sense that no meaning can be assigned to them with respect to the intended interpretation of product digraphs.

The **process plan** describes the manufacturing operations which are needed to obtain an output component from input components, specifying, where necessary, the order in which these operations are to be performed. In some cases, one operation must precede another, but in other cases, the order is not determined. The advantage of a concurrent model of computation is that it allow us to defer the determination of a fixed order of execution.

Another stipulation is that every operation in a process plan should be an **atomic operation**, that is, an operation which, once begun, will not be interruptible and will be allow to terminate naturally. The primary engineering justification for this stipulation is that in the general case it is either very difficult or impossible to determine the state of the device and the workpiece in sufficient detail to restart an arbitrarily interrupted operation. In practice, this is less restrictive than it might appear to be since atomicity put upper bounds on the length and complexity of process plan operations. Of course, there isn't, and one should not expect there to be, any formally precise criteria for identifying those operations which are just long enough and just complicated enough to satisfy the atomicity stipulation. Atomicity is a regulative ideal whose practical effect is to require that all properly terminating unit operations terminate with the devices in a well-defined state.

Another stipulation is that every product has at most finitely many components. This stipulation seems too obvious even to need statement, but the issues involved here are subtle. The digraph representation of products is consistent with the fact that there is no inherent upper bound on the number of possible component that can be identified. That is, from a purely conceptual point of view, there is no finest resolution of a product into components unless there is a finest resolution of the admissible operations in process plans. Suppose that a vertex X is supported by vertices A, B, Y, and Z; and for definiteness that the process plan consist of exactly three sequential steps: the first referencing A and B, the second referencing Y, and the third referencing Z. This configuration would permit several the development of a modified product digraph such that original vertex X would be replaced by two new vertices, say *X.first* and *X.second*, such that the process plan of *X.first* would consist of the first step of the process plan for vertex X (and would therefore reference A and B), and the process plan of *X.second* would consist of the final two steps of the process plan for vertex X. To finish the revision of the product digraph, we would need to make suitable syntactic modifications to the new process plans, assign distinct component identifiers to the new vertices, and then add an arc from *X.first* to *X.second*. Even though the revised product digraph will yield the same output as the original one, the structure of the product digraph and the component list of the product has clearly changed. If there were process plans which could not be further refined, then since all process plans are compositions of (possibly concurrent) operations, this would imply that the operations in such process plans would be irreducibly primitive. Even though the existence of irreducibly primitive operations is highly improbable, the construction just given does not rule this difficult case out. Fortunately, the component finiteness stipulation only requires that at any stage in the resolution of a product into its components, there are at most finitely many components, which leaves open the issue of the existence of conceptually primitive operations and maximally refined process plans.

A product digraph may be **capacitated** by assigning integer values to its arcs as follows: first, assume that one output component is to be produced, then determine how many units of each input component are required to produce that unit of output and assign that value to each of the corresponding in-arcs.

As mentioned above, manufacturing systems require another set of objects, to be called tool objects, which are not components of end products. Tool objects

include jigs and fixtures, cutting tools, dies and molds, gages, probes for inspection systems, and so on. Since tool objects are not product components, they do not appear as vertices in the product digraph. Tool lists may appear as an element of the contents of a component vertex. The use of tool objects, like the use of support components, must be fully specified in the associated process plan.

Plant Domains

Following Canuto [5] and Anwar and Nagi [2], plants, and their constituent devices, are subject to the logic of material flow. In each case, a well-defined collection of objects is presented as input, and, after a suitable delay, a well-defined collection of objects is delivered as output. Each device, when properly tooled, can perform any of a large collection of operations, and in general, there are three broad families of manufacturing operations applied to components: **material handling (transport)**, **fabrication**, and **assembly and joining**.

Every component has at least one path through the plant. The general characteristics of this path are fixed by the dependencies of the manufacturing operations on one another, that is, for every operation, we know all of the operations which must be finished before it is allowed to start. The detail path through the plant is determined by the scheduling algorithm which defines the actual devices to be used and order up the transportation needed to pre-position needed objects. The scheduling problem is difficult because it has to take into consideration the future scheduling of devices which at he time at which the schedule is being prepared are busy with other work. This requires a very accurate measure of the amount of time that a specific piece of work will take, which is the time standard for the operation.

Since plants are subject to a material flow discipline, **storage units** are critical elements of the plant domain. Every storage unit has a well-defined location in the plant, and, between operations, contains a fixed object bundle. An **object bundle** is a structure which associates a quantity to all component and tool objects. Bundles may be represented as arrays, but most bundles will be sparse, that is, most objects will be assigned a zero value. These sparse arrays can be compactly represented as lists of object/quantity pairs.

Storage units may be manipulated by two constitutive operators: the **draw** operator which reduces the on-hand quantity by the amount specified by the object bundle to be drawn; and the **delivery** operator which adds a new object bundle to the objects on-hand at the storage unit. There are two conditions which cause fatal errors: (1) an attempt is made to draw more of any object that the storage unit contains, and (2) an attempt to deliver more of any object than the storage unit can hold.

> **Df.** A **plant domain** is a set of digraphs whose vertices represent storage units and whose arcs represent manufacturing operations. (1) An arc from vertex A to vertex B indicates that an object bundle, drawn from storage unit A, is input to the manufacturing operation associated with the arc and the output object bundle is delivered to storage unit B. (2) The contents of every vertex consist of a unique storage unit identification symbol, a plant location, and an on-hand object bundle. The contents of every arc consist of a set of manufacturing operations. A transportation arc is one whose set of operations consists of object bundle transfers from one storage unit to another. A fabrication arc is one whose set of operations consists of processes that modify the form or material condition of the component workpieces. An assembly (joining) arc is one whose set of operations consists of assembly and joining processes. (3) Manufacturing operations are reduced to pairs of object bundles. Each bundle pair consists of an input bundle and an output bundle. An operation transforms the input bundle into the output bundle after a suitable lag.

A clearer understanding of structural characteristics of storage units may be obtained by resolving then into sets of smaller units, which will be called bins. Bins are either full, empty, or ready (non-full and non-empty). A storage unit is full or empty just in case all of its bins are. There are many variations on this theme. For example, bins may be assumed to be indistinguishable, that is, any object can be placed in any bin. Or bins may be assumed to be mutually distinct, that is, some objects can be placed in some bins but not others. The second case makes the definition of the effective state of a storage unit more complicated than the simple "one bin" case noted above. The fact that a storage unit could be full with respect to some objects and empty or ready with respect to others would obvious complicate the handling of the draw and delivery operators. The introduction of bins allows us to raise important questions. For example, can bins be shared by different storage units? A manufacturing system based on the "shared bin" principle can be built, but this would complicate the transportation model, compare [6]. Shared bins require a semaphore to control access: a shared bin can be accessed by a storage unit only if no other storage unit is accessing it, thereby locking all of the others out.

Semaphores are known to cause serious livelocking and deadlocking problems.

The proposed reduction of manufacturing operations to bundle pairs needs further explanation. Suppose that the collection of all component bundles has been defined for some arbitrarily selected product digraph. Then given any plant digraph, some bundle pairs will be feasible with respect to that plant digraph, but most will not. A component bundle pair is feasible if and only if there is a device, or a series of devices, however configured, which, given the input bundle, and any appropriate list of tools, could produce the output bundle; otherwise the bundle pair is infeasible.

The feasibility issue is an important one. The feasibility of bundle pairs determines the feasibility of the product digraph. Suppose that X is a vertex, then use the product digraph to collect all of the input components upon which the manufacture of X depends. This process defines a unique bundle pair, namely, one whose input bundle consists of the input components, and whose output bundle is the X component itself. It can be then be determined whether or not this bundle pair is feasible. If every vertex is feasible, then the product digraph as a whole is feasible.

This representation does not illuminate the deep structure of manufacturing operations and the devices that produce them. Suppose that the device in question is a CNC mill, then the most general representation of it is to define a coordinate frame of X, Y, and Z axes and to define a rectangular region, which is called the working envelop. A generalized cylinder may be used to represent the cutting tool. Because of the radial symmetry of the cutting tool, there will be a centerline of rotation for the tool and a point on the centerline which indicates the tool's nominal position in the machine tool's coordinate system. If the machine system is represented as a rigid body, then in a completely general way, the mechanical motion of the mill and its full set of operations may be defined in terms of the geometric motions of the tool as constrained by the mill's working envelop, together with a handful of auxiliary motions such as the operations used to turn the coolant on and off, to define the tool speed and feed rates, and so on. The kinematic motions of the mill are composed of translations and rotations, calculated with respect to defined axes of rotation. The mathematics of this model is straightforward. A program for such a device may be represented as a system of parametric equations defining the instantaneous settings for each translational and rotational axis. Even if force is neglected, there are a huge number of possible paths. In fact, so long as we concentrate on abstract devices about which rigid body assumptions can be made, it is possible to reduce the behavior of any device of this kind to a class of velocity functions which result in curves which are continuous and piecewise differentiable in the region of interest. But actual devices introduce much more complicated parameters. For example, physical devices have many more constraints: there is always a limit on the torque that an axis-drive motor can deliver as well as a limit on the smoothness of the acceleration function. Thus, many mathematically possible paths cannot be realized in practice. If the rigid body assumption is dropped, especially if we consider the deflections due to the loading the cutting tool, the complexity of the mathematical situation increases dramatically. The empirical index of the combined effects of all of these complicating factors must be ultimately derived from the measured deviations (error signals) of the actual path of the cutting tool from the nominal path, and possibly the projection of the error signals onto the translation and rotational axes. This discussion suggests that the production of a complete and exhaustive list of operations for any industrial device is impossible.

Discussion

This paper has sketched a framework for analyzing issues that are practically important in the context of manufacturing systems for plants. If this outline is even approximately right, then the work which remains to be done is a matter of providing a deeper and more rigorous elaboration of the computational theory sketched above and an equally thorough experimental investigation in order to determine whether or not the proposed theory is empirically valid.

Were this a purely theoretical exercise, that would be enough. But it isn't. What happened to the humans? It would be foolish to represent humans as items of capital equipment, but some place has to be made for them in this theory. The plant is a social world whose members are linked together in complicated patterns of mutual interdependence. A great number of skills are needed to bring technologically complex products to the market. Manufacturing can also be interpreted as a matter of correctly ordering the jobs such that each worker gets all of the information needed to make the appropriate decision, and then passes the intermediate components to other downstream workers. More precisely, the workers are linked together by channels through which work packets flow. This forms yet another digraph, but one in which the flow of packets from worker to worker may happen in both directions, but has a preferred, or normal, direction. In this model, the work flow follows the same routine. Each worker has an in-basket into which packets may be placed by upstream workers. The packet is opened and the job determined. If the worker is able to do the job, then he

begins it immediately and continues until either the job is finished or some unrecoverable error is encountered. In the first case, one or more new packets are dispatched downstream. In the second case, or if the worker is not able to do the job, then the packet is bumped upstream.

Learning is built in to this model, if we assume that workers can recall useful bits of their own past. Thus, if a downstream worker bumps a packet back because he is not able to do the job, then it is probably pointless to dispatch a similar packet to him again. But self-correcting systems of this sort are fertile grounds for job specialization. No matter what initial distribution of the packets is made, even a random distribution, under mild assumptions about the connectivity of the net, it will eventually happen that each packet will fall into a reasonable channel and flow downstream. This kind of system tends to be highly incompressible. If a plant were pushed to reduce its overall time to market, it has been demonstrated conclusively, and repeatedly, that a "work faster" methodology has no significant beneficial effect. In fact, it usually has the insidious negative effect of forcing each worker to search for a local optimum, which is one in which the burden is shifted either upstream or downstream. A much better alternative is to increase the amount of information available at each stage and correcting errors as quickly as possible, that is, to work smarter.

References

[1] Abelson, H. and Sussman, G.J. *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[2] Anwar, M.F. and R. Nagi, "Integrated scheduling of material handling and manufacturing activities for just-in-time production of complex assemblies," *International Journal of Production Research*, vol 36 (1998), 653-681.

[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. "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.

[5] Canuto, E. "Manufacturing Algebra," *Intelligent Automation and Soft Computing*, vol 2(1996), 389-406.

[6] Hoare, C.A.R. *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[7] Verganti, R. "Order overplanning with uncertain lumpy demand: a simplified theory," *International Journal of Production Research*, vol 35 (1997), 3229-3248.