Designing Programming Languages for Manufacturing Cells

David Alan Bourne
Paul Fussell

CMU-RI-TR-82-5

The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

April 1982

Copyright © 1982 Carnegie-Mellon University

This research was sponsored by the Robotics Institute, Carnegie-Mellon University, and, in part, by Westinghouse Corporation.

Abstract

A manufacturing cell is a complex collection of machines and electronics which must be intelligently supervised. It should be flexible enough to readily adapt to different part styles and robust enough to operate without human assistance for reasonable periods of time. A programming language with a correct choice of language properties can make meeting these demands and others like them a manageable programming task.

A rule based language in conjunction with a set of grammatical constraints supervises a cell which manufactures turbine blade pre-forms. The non-procedural nature of the language provides considerable flexibility in the operation of the cell. The rules are executed in no particular sequence, but rather as the cell is ready for them. Unfortunately, the non-procedural approach allows for unplanned interactions between rules. Most of these interactions can be avoided by defining which rules can operate concurrently. The language is logically linked to the pre-form cell through a simple database management system. The database system maintains a model of the cell used by the language interpreter to decide which rules to execute. This database system also protects the cell programmer from the low level programming details (e.g., communication protocols).

Table of Contents

1. Introduction 1
2. Language: Demands and Suggestions 2
    2.1. The Language Task 2
    2.2. The Program Forms 3
    2.3. Levels of Abstraction 4
    2.4. Language Primitives and the Definition of Truth 5
    2.5. Program Decomposability 5
    2.6. Error Detection: Pragmatics to Semantics to Syntax 6
        2.6.1. The Conflict Set 7
        2.6.2. Some Constraints 7
        2.6.3. Warnings 8
3. The Manufacturing Cell: System Architecture 10
    3.1. Cell Language Function and Structure 10
    3.2. Cell Equipment: Machines and Controllers 12
    3.3. Language Database 12
    3.4. Rule Execution Example 15
4. Summary 16
5. Acknowledgments 17
References 17

List of Figures

Figure 2-1: Illustrating a cell control hierarchy 4
Figure 2-2: Rule Database and the Conflict Set 7
Figure 2-3: Two robots contending for the same space 8
Figure 2-4: Robots with tails 9
Figure 3-1: Plan view of the Manufacturing Cell 10
Figure 3-2: A diagram of the control flow within the cell. 11
Figure 3-3: An example of interaction between the language interpreter and the equipment database. 14

1. Introduction

A manufacturing cell is a logical unit of machine tools combined to increase their utility. Justification of this cellular approach is provided by the speeded flow of parts, inventory control, reduction of labor costs, and the reduction of change-over time from one part style to another [2]. The coordination of the machine tools within the cell requires a sophisticated supervisory program managing each of the machine tools both alone and together. This involves , for example, the cold start of each machine tool, externally triggering its primitive programs, monitoring error states, and scheduling preventive maintenance. A sophisticated supervisory program will operate the cell unattended.

Manufacturing cells do not comfortably fit within the traditional classifications of computer controlled machine tools, *e.g.*, Computer Numerical Control (CNC) and Direct Numerical Control (DNC). The tendency in industry is to describe CNC as using a dedicated computer to control a machine tool and DNC as using a computer to distribute part programs to remote NC tools [5]. These definitions do not accurately describe the functionality of a flexible manufacturing cell, though some researchers extend the definition of a DNC controller.

A supervisory computer could operate the machine tools in a fixed sequence much as a NC machine tool operates. However, a multi-million dollar unmanned manufacturing cell must be managed rather than just operated. A cell management system sends its commands to the machine tools as each tool is ready rather than forcing a fixed and unalterable sequence on the entire cell (i.e., scheduling). It monitors the progress of the cell and either performs or schedules maintenance operations as they are required. It furthermore acts as an interface between the working machines and the high level factory operations (e.g., part design and purchasing). The final supervisory program is written in a rule based language whose run time system takes on the management responsibilities of the cell.

The evolution of programs designed for cell supervision is benchmarked by work such as that of Popplestone [8], Week [11], and McDonnell-Douglas [6][1]. These approaches to cell supervision have been based on procedural languages like FORTRAN, Pascal, APT (Automatic Programmed Tool), or a modification of these. Popplestone and McDonnell-Douglas based their work on a superset of APT (Robot APT and MCL, respectively). Week's work was done in a combination of FORTRAN and CAMAC (hardware and software protocols for low level communication).

Procedural languages have been designed around a Von Neumann computer architecture which has a single processor and a single memory store. This architecture sets up a bottleneck in the flow of instructions [1], because each instruction is expected to go through the processor in turn. A procedural language design, and a single processor computer architecture are not appropriate for the tasks of a cell management system. Each machine tool is equipped with its own controller which manages the machine in real time. These controllers are in turn linked to a central processor. A non-procedural language describes an algorithm in such a way as to make the order of program statements unimportant. In fact, the statements are executed as some goal process needs to invoke them, such as keeping all of the machine tools busy. In the case of a manufacturing cell, the instructions take a long time to execute because they represent physical actions. Therefore, a non-procedural interpreter is able to direct a flow of high-level instructions to each controller as they are needed. The discrepancy in time, between computational and physical actions, avoids the bottleneck in the central computer.

---

[1]The McDonnell-Douglas work is sponsored by the Air Force Integrated Computer Aided Manufacturing (ICAM) Project.

A rule based language is essentially a set of conditionals which can be treated non-procedurally. The condition amounts to the pre-conditions for executing a particular cell instruction (e.g., "is the furnace door open? -> then get part"). This implies that the supervisory program is able to sequence the events within the cell even though the events do not occur in a rigid order. An interrupt driven system is also non-procedural in nature, but it obscures the conditions of execution making the final system difficult to understand.

A formal deductive system is the primary example of a non-procedural language. The order in which theorems are proved has no effect on the set of provable sentences. That is, there are no relevant side effects in the the course of a proof which impinge on the system as a whole. Unfortunately, systems which have not been adequately formalized are characterized by many unsuspected side effects. These side effects are manifested by rules which interact because of their interdependence. Two robots could be put in a collision course under one ordering and affect useful work under another. A set of grammatical constraints is proposed which filters out rules that have undesirable side effects in a particular context.

Our research is directed at the theoretical aspects of language design as well as the practical matters which are involved in implementing a cell. The cell is currently being installed at the Westinghouse Electric Corporation, Turbine Components Plant in Winston-Salem, North Carolina. It will produce steam turbine blade pre-forms from billets (cylindrical metal stock). The major mechanical process in the cell is open-die forging which operates on billets that are in excess of 2000° F. The cell contains an industrial rotary furnace, two materials handling robots, an open-die forge , a vision based loading station for acquisition of the billets and a vision based gaging station for inspection of the forged pre-forms. The goal of the language development is to provide a system capable of autonomously supervising the cell operation for a reasonable period of time, *e.g.*, a weekend.

There must be a logical connection between physical machines in this pre-form cell and the words in the language. That is, the terms in the language must be meaningful to the supervisory program. A simple database management system makes this association between words and salient cell features. The features are contained in a highly structured database and are used to model the cell at a particular time. Whenever the language interpreter accesses the database through the management system some of the cell features are updated.

The rest of this paper is divided into two major parts. The first part defines a language which is based around a set of formal properties which are useful for cell programming. These properties guide the language definition through the sub-sections. And the final part of the paper clarifies the connection between the characteristic language properties and our particular manufacturing application.

2. Language: Demands and Suggestions

2.1. The Language Task

A complex manufacturing cell is composed of many computer controlled tools. These may include robotic arms and machining equipment or sophisticated sensors such as vision systems. The best return on capital investment for this expensive equipment is realized by optimizing the mean throughput time. It is not important to save milli-seconds by optimizing the moves of the robots, but it is critical to maintain the flow of the parts within the cell and prevent untoward shut-downs of the cell due to equipment failures such as dirty

oil filters. The cell is to be unmanned and should be capable of preventing hardware failures by automatically scheduling tasks for preventive maintenance. This is achieved by placing sensors on the mechanical hardware, by providing the means to convert the sensor information into sensible numbers and by having *a priori* knowledge of when the hardware is likely to fail.

Another means of optimizing throughput involves the ability to easily decompose a program into parallel sections that can execute independently. Each machine tool typically takes several minutes to perform a single operation making parallel execution of independent operations important.

The resulting program is complex and needs to be easy to update. A program is modified when new part styles are manufactured and when the cell itself is altered as in the addition of new machine tools. One way to manage the complexity of the program is to structure it in a way that it can be understood at different levels of detail. And, of course, the final program should be if at all possible error free.

In order to achieve these demands the language must possess a number of formal properties which make the problems inherently manageable. Our language design is based on such a set of formal properties which can be justified independently of the particular language. We feel that the point here is not to design a new language which is meant to be every programmer's panacea, but rather to encapsulate a special set of language properties under one framework.

2.2. The Program Forms

A manufacturing cell is usually made up of a set of machine tools each of which has many functions and error states. By establishing the pre-conditions of each machine function, it is possible to execute program segments as they are needed rather than as they appear in the procedural flow. In addition, most machine error states are paired with a suggested course of action. Both of these basic requirements call for a non-procedural rule based language. This programming paradigm has been extensively studied under the guise of **production systems** and has been reviewed by Waterman and Hayes-Roth [10].

The general form of a program and its statements is a conditional.

(antecedent) → {consequent}

The left hand side is evaluated as a boolean expression, and if **TRUE** then the right hand side is executed as a set of sequential actions. An example program segment which commands a robot to place a work piece **(Billet)** in a furnace could be written in the following way.

```
(And  (Hold Billet)              { (Move Robot Door)
      (Open Door)          →      (Move Robot Furnace)
      (Vacant Space))             (Place Billet)
                                  (Move Robot Door)
                                  (Close Door) }
```

The convention used here is that the first element[2] of a list on the left hand side of a rule is a truth predicate

[2]The convention of putting a function first in a list is compatible with the LISP programming language.

and the first element of a list on the right hand side is a command function.[3] For example, the truth predicate **Hold** is TRUE if and only if the robot is actually holding a **Billet.**

2.3. Levels of Abstraction

Structuring a program hierarchically has many different advantages. The entire program can be seen and understood at a glance even though it may be at a coarse level of abstraction. This convenience is available to everyone who needs to look at the program, including the shop foreman, the programmer, and even the computer doing the execution. Finer levels of abstraction provide more and more details about particular aspects of the manufacturing cell. This makes it both easy to find details in an existing programs and to add new rules without disturbing an existing program's internal structure.

In keeping with the program forms discussed previously each program is made up of a single conditional: the root of the program.

(Active Cell) → {consequent}

The consequent in turn can be a set of conditionals or a set of basic actions. This rule makes it possible to turn the whole cell on or off by making the boolean **Active** of **Cell** TRUE or FALSE respectively.

The next level of abstraction naturally falls into the different modes of operation for the cell. These modes of operation might include: cold starting the cell, scheduling preventive maintenance and the basic execution cycle of the machines. The final levels make up the actual cell control, and its verificational sequence.

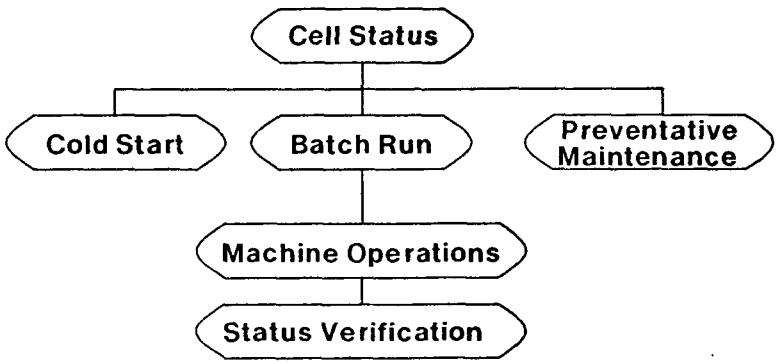


Figure 2-1: Illustrating a cell control hierarchy

[3] The notation for implication (i.e., "→") is only used to highlight the difference between the left and right hand sides of a rule. It could just as easily be written as a function found first in the list (i.e., "(if (antecedent) {consequent})").

2.4. Language Primitives and the Definition of Truth

The language primitives correspond to primitive cell functions and states. The primitive cell functions are each a set of non-decomposable machine actions which return TRUE upon successfully completing the task and FALSE otherwise. The machine states can also be viewed as a function which declares that the machine is in a particular state. If the machine is already in the correct state then TRUE is returned and FALSE otherwise. These simple definitions constitute the basic semantics of the language.

The cell primitives are a set of functions from which complex programs can be built. Two example primitives would be a robot move command and a particular machine location. If the machine position changes,[4] then the only change that has to be made to the language is to the semantical definition of the term, and not to the program itself.

A cell primitive has several important characteristics.

- A primitive is a simple function which never needs to be decomposed further within a particular manufacturing cell.
- A primitive is useful and therefore used in complex configurations. In other words, a manufacturing cell for making turbine blades would not have a primitive which returns the current phase of the moon.
- A minor change in a primitive does not disrupt the portions of the supervisory program which use it.
- The details of how a primitive accomplishes its task are not needed by the supervisory program.
- The implementation of the primitive is confined to one machine within the cell. This corresponds to a practical restriction which forces the real time control of a machine to be contained within its own controller.

None of these characterizations of a primitive are necessary and in fact are only used as guidelines to the overall system design. A sample set of primitives is given for the open-die forging cell in Section 3.

2.5. Program Decomposability

Decomposing programs into independent segments is an extraordinarily difficult problem for programs which are written in an ordinary procedural language. However, programs written in a rule based language are claimed to be made of independent chunks which can be executed in any context. That is, they are already decomposed into independent pieces. This is a very strong condition, because it is so difficult to guarantee that a rule's execution has no side effects to the surrounding environment. This problem is especially acute in physical systems such as a manufacturing cell where some side effects may not have been taken into account within the model of the cell. This incompleteness is generally due to the programmer's ignorance of subtle iteractions between machine tools. The effect of interactions between rules has also been observed by other researchers [7].

---

[4] Only static locations would be made primitive. For example, the position of a furnace door would be primitive, and its position could be changed only after a major construction project.

2.6. Error Detection: Pragmatics to Semantics to Syntax

A programming error in a manufacturing cell easily could cause a several hundred thousand dollar accident. A 100-lb. work piece could be dropped on a laser etching device or a furnace could fail to open its door before a robot tries to enter it. There are many different kinds of errors that amount to a miscommunication between the machine and the programmer. That is, the programmer does not always say what he means. Many of these errors can never be detected, but many of the outlandish errors can be detected and then avoided using linguistic techniques.

Program errors can be found using a range of different language mechanisms. At one extreme you can sit back and wait, and watch the machines crash into one another. This is a pragmatic approach. It is an effective but expensive means of error detection. A more reasonable approach would be to simulate the program using a computer model (i.e., semantics) of the physical machines in the cell. The program execution would then cause the model to go through its paces. For example, two polygons intersecting might indicate that if this program were to run using real robots, then they would collide. Barry Soroka at Stanford has recently used this approach to help him debug robot programs [9]. Unfortunately, an accurate simulation is computationally expensive and is only as reliable as the model is accurate. It should be pointed out that a purely graphic simulation does not offer ANY error detection facilities *per se*, but rather is only a tool for the programmer to see his own errors. It is certainly possible to extend the simulation to include geometrical constraints that prohibit graphical primitives from intersecting, but this is at best a first cut at the possible errors since it says nothing about a robot which needs to instantaneously stop. It would then be possible to add deceleration constraints to all the robot movements, but then this list of constraints can be extended *ad infinitum*. In fact, the general notion of constraints can be extracted and used to modify the basic syntax of the final programming language. This is the final step in the continuum of solutions. More elaborately, programs that would cause machine collisions with an unconstrained language would be meaningless and would therefore never get to the execution phase. The syntactic approach to error detection has two advantages over the more traditional means of simulation. It provides a more streamlined means of encoding the real world constraints, and it directly prohibits a programmer from writing the programs in the first place.

The real world constraints of a manufacturing cell can be embedded in the grammar of a language. For the moment, consider a manufacturing cell with a single robot. Such a cell has many of the complications that are found in cells containing 15 machine tools. The most obvious commands for a robot include a move instruction that is constrained by the physical capability of the robot, at least in terms of position and speed.

(Move x y z speed)

One sure way to avoid problems is to make the robot movements primitive, so that the robot arm always is accelerated and decelerated properly. This is in opposition to making both positive and negative acceleration primitive. Unfortunately this is no solution at all, since when the rule of making programs primitive is applied ubiquitously, it trivializes the idea of having a language. The entire cell program is made into one huge primitive. The alternative is to find rules which allow robot primitives to be combined in complex ways, while avoiding the disastrous combinations exemplified by robot collisions.

2.6.1. The Conflict Set

At each level of the hierarchy at a particular time there is a set of satisfiable rules; the set of rules whose antecedents are TRUE. This set is called the conflict set. The name originated on a sequential machine where it was necessary to choose which rule could be executed first. In this case each rule is sent to a physically different machine, so the rules can be executed in parallel.

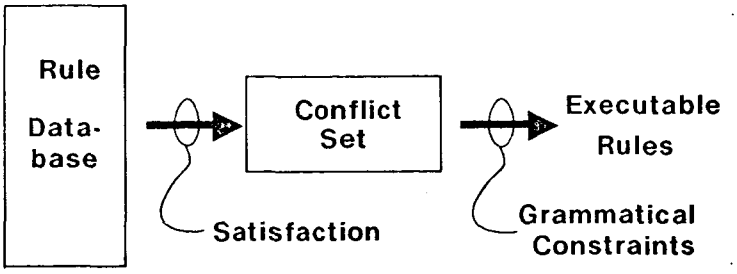


Figure 2-2: Rule Database and the Conflict Set

2.6.2. Some Constraints

Grammatical constraints are used to restrict some rules in the conflict set from being executed. These constraints examine the right hand sides of the conflict set rules and determine which predicate argument sets are incompatible. The rules passing this examination are executed while those rejected are deleted from the conflict set. One simple constraint prohibits the host computer from sending more than one set of instructions to a particular controller.

A → {(Robot-1-gripper open)}
B → {(Robot-1 gripper closed)}

In this example only one of the rules would send its command to the the **Robot-1-gripper** controller.

Figures 2-3 and 2-4 show two robots that can reach into each other's range. The next constraint restrains the two robots from working in the critical region at the same time.

A → {(Robot-1-move D)}
B → {(Robot-2-move d)}

The grammatical constraint must take several factors into account in order to determine that these rules are incompatible. Each of the discrete points {C,c,d} should be marked as being part of the critical region. Therefore, this constraint prevents both of the robots from going through the critical region in the same system cycle. One approach is to encode the robot movement points in such a way that movements can be defined in terms of intervals. For example, a lettering scheme makes it possible to represent a movement from point A to point D as the closed interval [A D], which implies that the points B and C are included within the interval. This suggestion makes it possible to test the constraint relation with a subset operation.

IF Robot-interval-1 ∩ Critical-set and
Robot-interval-2 ∩ Critical-set
THEN constrain a rule [5]

Several interesting problems arise when a robot can take two paths to a particular point. One path could be short and intersect with the critical region and another path could take the long way around. The system must then decide whether it is worth waiting for a path through the critical region or whether it should just take the alternate route. Fortunately, it is still easy to represent the circular nature of a robot movement in an interval notation by reversing the arguments (e.g., [D A]).

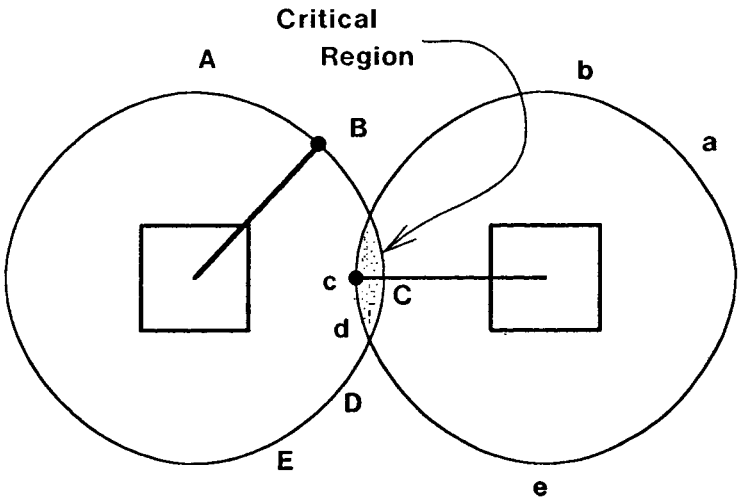


Figure 2-3: Two robots contending for the same space

The tailed robots in Figure 2-4 represent a difficult class of problems, because they strike at the heart of the inadequacy of abstract models. Moving the robot from point A to point B seems to be a perfectly reasonable action for the head of the robot, while the move has disastrous effects on the tail. While the tail can be dealt with in the same way and in conjunction with the head, sooner or later something will be left out of the model. A hydraulic hose may extend into the critical region in some robot positions and not in others. Perhaps a more convincing example of this dilemma is a robot moving different size parts. This actually changes the size and shape of the critical region and depends on how the robot is holding the piece.

2.6.3. Warnings

The constraints are making up for programs that are incompletely specified. For example, the ordering of program statements is intentionally left open until run time. This incompleteness is the source of some trepidation.

If the conflict set ever became too large, it would be time consuming to check all of the combinations that

---

[5] A non-nil intersection is intended to return **TRUE**

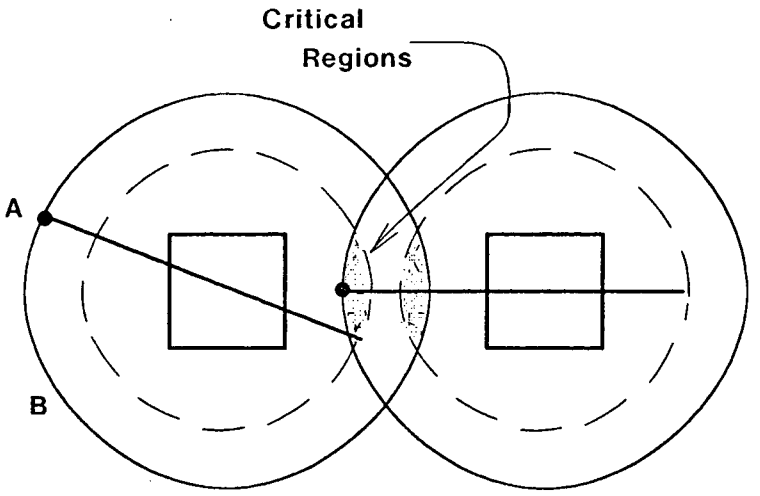


Figure 2-4: Robots with tails

would be called for by the constraints. This would only become a problem in an large manufacturing cell, because the conflict set size is bounded by the number of controllers. This may become a determining factor in drawing cell boundaries.

A rule may be continually thrown out of contention based on the constraints alone. There are two possible resolutions to this problem. First, the rule may be naturally executed as the competing rule completes its activities. And second, this could be directly prohibited by the system forcing the rules to alternate.

Constraints not only have to consider conflicting rules, but they must consider the current state of the cell. This problem is brought on by the hierarchical rule structure which by changing the cell at one level could cause undetectable errors at another level. The constraints must be given access to the database in order for them to make the appropriate checks between the rules and the cell state. The only other option would be to save an entire list of executed statements and to recalculate the cell state. Since this option is absurd, the purity of the grammatical constraints must be violated by giving them semantical access.

The system may reach a state of deadlock. That is, where there are no satisfiable rules left to execute because of resource contention. Consider a robot which is waiting for another robot to get out of its way. The state of deadlock occurs when the second robot is also waiting for the first robot to get out of its way. This assumes that they are clever enough not to run into each other in the first place. The problem occurs in essentially every kind of operating system and has never been solved to anyone's real satisfaction. Two approaches have been to avoid the problem through the use of semaphores, and to pre-empt one device in favor of another once the state of deadlock has been detected. Viewing physical space as a resource, it is possible to assign semaphores to the critical regions. The semaphores represent semantical access for the grammatical constraints and thus address the robot deadlock problem.

A systematic approach is being developed to detect and resolve each of these problems. However, these solutions will only emerge after extensive research and an enormous amount of experience has been gained with an implemented system. The implementation, and its relationship with the rule based language are the focus of the next section.

3. The Manufacturing Cell: System Architecture

3.1. Cell Language Function and Structure

This language is implemented for a cell producing turbine blade pre-forms. The production of these pre-forms is the first step in the production of one family of turbine blades which are used in steam turbines. Pre-forms are produced by open-die forging of cylindrical billets. Additionally, the cell heats the billet, crops the pre-form and stamps the pre-form with appropriate model and batch numbers. Finally, the pre-forms are gauged using a computer based vision system. Both the billets and pre-forms are handled by two large robots. The parts flow from the loading racks to the furnace, through the forging machine, to the cropping/gauging station and finally to a pre-form bin. Figure 3-1 shows the circular nature of this material flow.

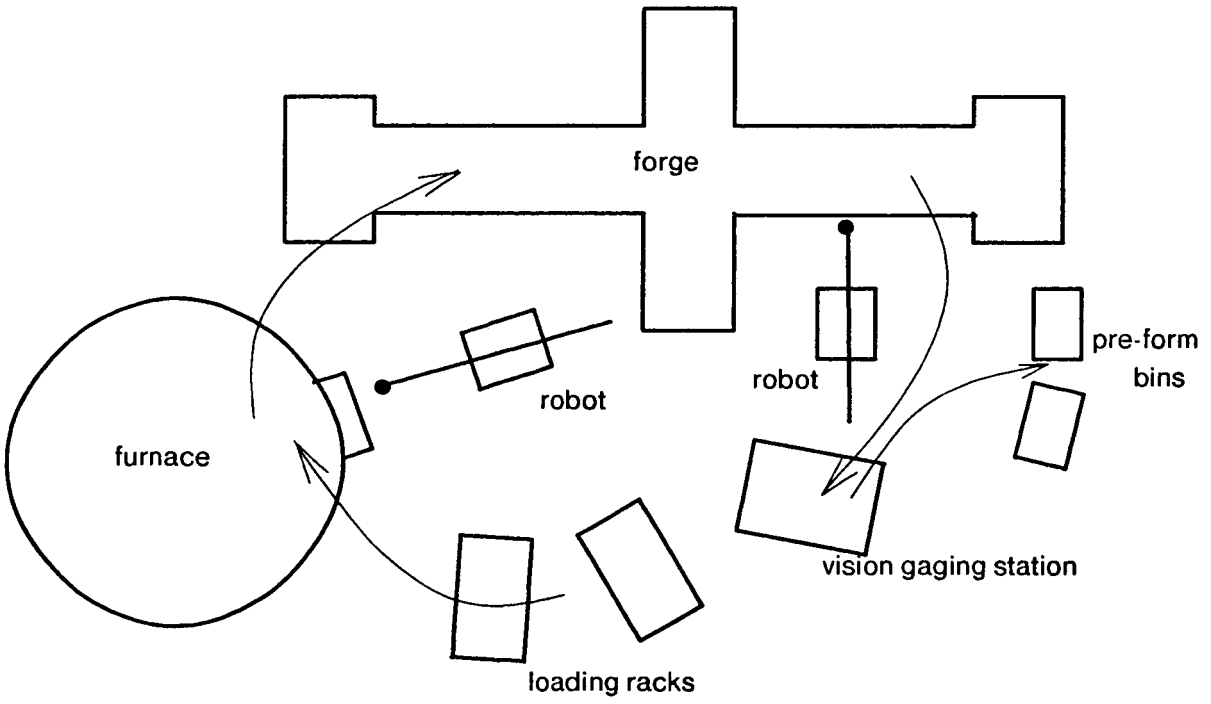


Figure 3-1: Plan view of the Manufacturing Cell

The program structure and its rules are mirrored in the physical construction of the cell. The top levels of the hierarchy control the management functions of the cell and the lower levels describe the interactions between the rules and the equipment. Figure 3-2 illustrates this linkage.

The rules interact with the physical cell by determining the truth value of antecedents and executing cell primitives. This interaction is through a database, as is generally the case with production systems [4]. The database manager, in part, provides the truth content of any particular antecedent or consequence to the language interpreter (including the highest level rule -- **active cell**). The database thus acts as the interface between the rule based language and the physical cell.

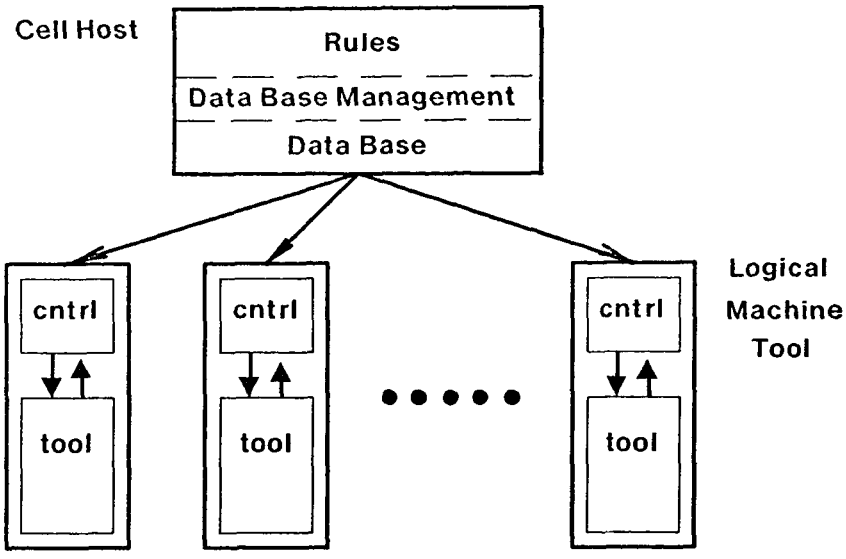


Figure 3-2: A diagram of the control flow within the cell.

The levels below the database consist of the machine controllers and the machine tools. This language views the controllers and equipment as logically one. This allows the supervisor to ask a controller to execute a program, but does not burden the supervisor with the details of controller operation. The fact that the controller did, or did not, execute the program is all that is relevant to the supervisor. Additional information is valuable for fault tolerance and maintenance, but supervision is achieved by viewing the controller and equipment as one logical unit.

Inter-machine communication within the cell is handled exclusively through the supervisory host. This includes both the cold-start of the manufacturing cell and the machine tool interaction during cell operation. During a cold-start the operating systems and the parts programs for each controller originate from the plant computer and are distributed by the host. Communication between operating machine tools frequently occurs during cell operation. For example, the communication between the robot and the forge is passed through the host. When the robot passes a billet to the forge, it must ask the forge, via the host, to close its chuck jaws.

Language status and maintenance operations interact with the database to provide cell operating parameters to the supervisor. The status operations are classified as providing information about parameters where the supervisor can have immediate consequence. For example, the robot grippers are equipped with strain gauges to indicate if a billet has been successfully acquired. Checking those strain gauges indicates to the supervisor the success of the acquisition and either fires a rule for corrective action or allows continued normal execution. In the same vein, the supervisor will have access to information concerning the status of the furnace atmosphere. If the atmosphere goes out of tolerance, a rule will be fired to correct it.

Maintenance operations are rules generally designed to provide information for fault correction and for the scheduling of preventative maintenance tasks. Typically the supervisor does not have immediate control over

the information provided by the maintenance rules. The robot servicing the furnace, for instance, should not shut down due to a filter clog with its end-effecter within the furnace. The supervisor can only prevent this from happening, however, by indicating that the filter should be replaced. The longer time response for correction of a problem indicated by a given maintenance rule suggests that they can be separated from the other rules within the language. In this way the maintenance rules generating a list of equipment needing service can be fired when the supervisor has time.

3.2. Cell Equipment: Machines and Controllers

The cell is programmed by creating sub-programs in each machine controller. The set of sub-programs within the controller constitute the possible actions of the machine tool for a given turbine blade batch run. The sub-programs also constitute many of the primitives of the cell language; the sub-programs are primitives for the supervisor. The primitives, as discussed above, are executed when a rule fires within the cell program. Thus the machine controllers must be capable of executing an internal program upon command from the cell supervisor. The cold start condition of the cell requires the cell host to pass parts programs to the machine tools, so they must have the capability of receiving previously written programs from the cell host.

To maintain an accurate correspondence between the supervisor model and the physical condition of the cell the supervisor must have access to timely information on the cell operating status. Items such as the current program any one machine is running and the exit status of the last program a tool ran are important. The progression of supervisor complexity toward autonomous operation for an extended period of time requires a more encompassing model. This model, in turn, requires more information from the cell. One of the most interesting uses for this additional input is the detection of faults within the manufacturing process by visually inspecting the finished pre-form [3]. The visual inspection is being done by equipment identical to that doing the billet location for the initial part acquisition.

The constraints of the controllers have important consequences for the language. The host computer initiates all communication between a tool and itself. The available controller designs, however, allow them to acknowledge messages only when they are ready. The robot controllers, for example, are essentially completely busy during robot translation. Only between the execution of sub-programs do they have time to correspond with the supervisor. This restricts the supervisor from terminating running sub-programs.

3.3. Language Database

The database system is the interface between the supervisor and the machine controllers. The information within the database is used to pass and obtain operating parameters to and from the machine tools. A software interface driver is implemented to execute this communication. The driver understands the protocol for the communication between the host and the machine tool while the database supplies the driver with the appropriate parameters for the task at hand. The database is constructed to provide a consistent format for those parameters. The cell has a large number of inputs to the supervisor, but these tend to be either linear functions of the physical variable or simple binary inputs. This has allowed us to construct the database around an element which varies from zero to one. This element represents, in a uniform manner, the value of any of the inputs from the cell. Previously known minimum and maximum values are included in the database as parameters to a normalization routine. The fundamental idea here is the ability to represent a varied and complex set of inputs as a uniform set of values. This internal value can then be easily translated to numbers appropriate for either an operator or a machine tool.

The database uses abstract data types which include both the data and the functions needed to model the equipment. An entry in the database will typically include the following classes of information considering the furnace as an example:

- The logical name of a software driver which understands the protocol for communication between the host and the machine controller[6].

- The normalized value of the state. This value will reflect the last sampled value of the state normalized to a range between zero (0) and one (1). An executing rule will cause a new value to be sent to the machine tool. The values in the database are then updated from the new machine tool state.

- The maximum and minimum values of the state as viewed by the program rules. These are numbers representing the values of the state as understood by the programmer, *e.g.*, furnace temperature in degrees Fahrenheit.

- The maximum and minimum values of the state as viewed by the machine controller. These values are related to the hardware on the controller, *e.g.*, furnace temperature represented by a number between 0 and 4095 for a hardware device such as a 12-bit D/A converter.

- The memory location within the remote controller from which the device is controlled.

- A number indicating how quickly the machine state changes. This information is used in deciding how frequently the state needs to be updated. The furnace temperature changes rather slowly with respect to the angular position of the furnace hearth. Therefore, the hearth position will be updated more frequently than the furnace temperature.

To capitalize on the advantages discussed above, this database structure will also be used for the other controller entries. The number representing the memory location within the remote controller will point to, for example, an executable robot sub-program. The robot will return a truth value (true for successful completion of the task and false otherwise) when the sub-program completes. This truth value will be placed in the state value.

Figure 3-3 shows an example of the interaction which takes place between the language interpreter and the equipment database. In this example the interpreter operates on a rule which opens the furnace door if the door is closed. The database manager (called "Get Truth" here) ascertains the truth content of a rule's antecedent. If true, the interpreter sends the consequence to the database manager and it is executed. The equipment database contains:

- the "name" of the furnace driver

- a normalized value of the door state (in this case, either opened or closed)

- minimum and maximum values (all of which are zero or one for this binary system)

---

[6]The logical name is connected to the address of the driver.

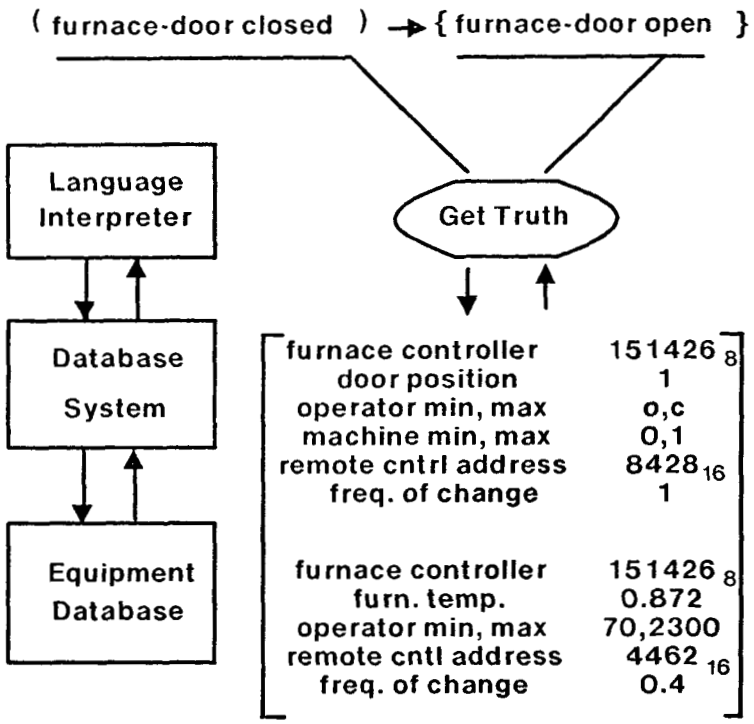


Figure 3-3: An example of interaction between the language interpreter and the equipment database.

- the memory location within the furnace controller that controls the furnace door.

The other entry in figure 3-3 is a representative entry for the furnace temperature control.

This portion of the database will also contain two other entries for registers showing the actual state of the door[7] . The combination of these two entries provide the supervisor with three possibilities of furnace door positions: open, closed or indeterminate. The indeterminate state is composed of two states, 'door neither open or closed' and 'door both open and closed.' The latter is a serious error, representing a failure in the cell.

The primary non-transportability of the system rests in the entries of the database. This implies that the movement of the supervisor to a different manufacturing cell would primarily require re-writing the database. By the same token, the majority of the work involved in adding a new machine tool to an existing cell is adding new entries to the database.

[7]The door open and door closed states are registered by limit switches on the furnace.

3.4. Rule Execution Example

A brief illustration of rule execution should help clarify how a program is written and executed. This example is concerned with a set of rules supervising a robot moving cold billets from the loading rack to the furnace. For simplicity, there are two 'staging' areas in this transfer. One is in front of the furnace and one in front of one of the loading racks. The staging areas are, in a significant sense, safe locations for the robots: they are not in any of the critical regions of the cell nor do they place any portion of the robot in a critical region. The robot in this example starts at the furnace staging area which is an endpoint of one of the robot primitives.

A

```
(AND  (Acquired Billet InRack)                  →  { (Locate Billet InRack) }
      (NOT(Located Billet InRack)) )
```

B

```
(AND  (At Robot LoadStage)                      →  { (Pass Rack BilletLocation ToRobot)
      (Located Billet InRack)                       (Acquire Billet InRack) }
      (NOT(Gripped Billet)) )
```

C

```
(AND  (At Robot LoadStage)                      →  { (Move Robot ToFurnaceStage) }
      (Gripped Billet) )
```

D

```
(AND  (Gripped Billet)                          →  { (Open FurnaceDoor)
      (Cold Billet)                                 (Pass Furnace BilletLocation ToRobot)
      (At Robot FurnaceStage)                       (Place Billet InFurnace)
      (NOT(Moving FurnaceHearth )) )                (Close FurnaceDoor) }
```

E

```
(AND  (At Robot FurnaceStage)                   →  { (Move Robot ToLoadStage) }
      (NOT(Gripped Billet))
      (NOT(Full Furnace)) )
```

The syntax of the rules is described in Section 2.2. To fully understand this example, a few comments on the semantics of the functions are noted.

- Tenses are used to distinguish between boolean functions and imperative functions. For example, **(Acquired Billet InRack)** is a boolean function which returns **TRUE** if and only if a billet has already been acquired. This is distinguished from **(Acquire Billet InRack)** which commands the robot to acquire a billet.

- **Acquire** and **Place** implicitly refer to robot sub-programs. Their arguments specify to the DBM (Data Base Manager) which sub-program should be executed by the robot.

- The **Locate** function in rule A tells the vision system to locate a billet in the loading rack.

- **Pass** explicitly 'passes' a value from one machine controller to another. For example, **(Pass Rack BilletLocation ToRobot)** passes the billet location which is found by the vision controller to the robot controller.

- **At** is a boolean function with two arguments. The first argument is a machine at the position of the second argument.

A trace of the execution might appear as follows.

- The language interpreter acting with the DBM determines that the rules **A** and **E** can be passed to the conflict set. The constraints will not reject these rules, so they are passed to the DBM for execution. The DBM will tell the vision system to execute its billet location primitive which returns the billet location. It will also instruct the robot controller to execute a sub-program that moves the robot to the loading rack staging area. The two separate consequences can execute simultaneously.

- The next pass of the interpreter discovers that no rules can be passed to the conflict set. This will continue to be the case until the consequences of rules **A** and **E** have completed execution. At this time, the interpreter will discover that rule **B** can be executed. **B** instructs the robot to acquire a billet from the loading rack. The **(Acquire Billet InRack)** consequence in rule **B** is a robot primitive which

    - moves the robot from the staging area to a location over the billet

    - lowers the gripper to the height of the billet

    - closes the gripper

    - raises the gripper

    - and returns the robot to the staging area following a pre-programmed path such that the billet does not contact the loading rack.

- After the consequence of rule **B** has completed, the interpreter passes the rules **A** and **C** to the conflict set. Again, the constraints will not reject either of these rules.

- At the completion of **C**, it is possible to execute **D**. Rule **E** is rejected by the interpreter because the gripper does contain a billet. If the programmer had forgotten the first consequence in rule **D**, the constraints would have rejected **D** from the conflict set, because execution of **D** would have placed both the furnace door and the robot arm in the critical region of the door.

- Finally, **E** can again be fired to move the robot to the loading rack staging area.

4. Summary

The manufacturing cell of the future is the basic unit of a flexible factory. If a cell is expected to do a wide range of tasks, then there must be a straightforward way of reprogramming it. This involves a programming language which encapsulates a set of properties that makes the programming task easy and which helps the programmer avoid costly errors. The same requirements would be found in a more sophisticated CAD/CAM system where the program would be automatically constructed from a part design.

A non-procedural language has been chosen as the most likely candidate for cell control. Unfortunately, the very advantages of this scheme, its non-sequential nature, also are the cause of its biggest problems. Unwanted interactions between program statements could be translated into actual robot collisions which must be avoided. Therefore, grammatical constraints have been added to choose which rules in the conflict set can be simultaneously executed. The implemented manufacturing cell will provide us with a unique opportunity for developing new and more powerful constraints which in the long term will support a more basic theory of language development.

5. Acknowledgments

The implementation of a manufacturing cell requires the dedication of many people. The authors wish to thank Paul K. Wright, and Jerry Colyer for much of the needed support and guidance. In addition this research would not have been possible without the support of the Westinghouse Turbine Components Plant.

References

[1] J. Backus.
Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs.
CACM 21(8):613-641, August, 1978.

[2] Oyvind Bjorke.
Computer Aided Part Manufacturing.
Computers in Industry 1:3-9, 1979.

[3] D.A. Bourne, R. Milligan, and P.K. Wright.
Fault Detection in Manufacturing Cells Based on Three Dimensional Visual Information.
In *Proceedings of the Robot and Vision Seminar.* to appear June, 1982.

[4] Davis, R., and King, J.
An Overview of Production Systems.
In Elcock, E. W., and Michie, D. (editor), *Machine Intelligence*, pages 300-332. Wiley, New York, 1976.

[5] Toepperwein, L. L., Blackmon, M. T., et al.
ICAM Robotics Application Guide.
Technical Report AFWAL-TR-80-4042, Volume II, General Dynamics Corporation, April, 1980.

[6] Ennis, G. E., Eastwood, M. A.
Robotic System for Aerospace Batch Manufacturing.
Interim Technical Report IR-812-8, McDonnell Douglas Corporation, 1979.

[7] David Jack Mostow, Frederick Hayes-Roth.
A Production System for Speech Understanding.
In D. A. Waterman and Frederick Hayes-Roth (editor), *Pattern-Directed Inference Systems*, pages 471-482. Academic Press, New York, 1978.

[8] R. J. Popplestone, A. P. Ambler, and I. Bellos.
RAPT: A language for describing assemblies.
The Industrial Robot :131-137, September, 1978.

[9] Barry I. Soroka.
Debugging Robot Programs With A Simulator.
CADCAM-8 Conference , November, 1980.

[10] D. A. Waterman, Frederick Hayes-Roth.
An Overview of Pattern Directed Inference Systems.
In D. A. Waterman and Frederick Hayes-Roth (editor), *Pattern-Directed Inference Systems*, pages 3-22. Academic Press, New York, 1978.

[11] Weck, m., Zenner, K., and Tuchelmann, Y.
New Developments of Data Processing in Computer Controlled Manufacturing Systems.
Technical Paper MS79-161, Society of Manufacturing Engineers, 1979.