

David Alan Bourne

CML: A Meta-Interpreter for Manufacturing

Introduction

The development of advanced robotics brought expectations of increased productivity and quality control, but to everyone's disappointment, these expectations still have not been realized. Advanced stand-alone machines have not greatly improved productivity, and integrating large systems has been prohibitively expensive. What is worse, the few integration projects that have been undertaken took inordinate amounts of engineering time. Several projects took more than 50 man-years to complete, and engineers spent most of this time trying to put round plugs into square sockets. Some machines were not designed to allow for any communication, and while others provided only partial communications that presume a person is operating the front panel. The few machines that were designed to be integrated into a system were rarely compatible, even within a single vendor.

At least one dream for factory automation should be simple: Roll a computer onto a factory floor; plug it into a set of machines from different manufacturers; start a program; and then with absolutely no traditional programming, begin the task of integrating the machines into a cooperative cell. CML brings us within reach of this dream. The pursuit of this dream demands that CML cope with a range of approaches that are encountered in manufacturing:

- **Generic versus Custom:** The software tools for building flexible manufacturing systems must be generic, so that the same tools can be used over and over again to solve new system integration problems. Also, these tools must be powerful enough to drastically reduce the development time of the overall system. As a rule of thumb, our lab has set a goal of increasing productivity for building automation software by a factor of 30 over the existing approach of customizing new applications.*

David Alan Bourne is a research scientist at The Robotics Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213

* The magic number 30 is in recognition of the fact that it usually takes about 30 days to implement an idea which takes one day to develop. In other words, the ideal situation would be to eliminate all of the time that is normally spent programming an idea rather than to eliminate coming up with the idea

- **Mixed versus Standardized:** What makes our approach different from most others is that we are building systems which work with existing equipment, making only a few minimal assumptions for example, about nominal communication abilities (Fussell, Wright, and Bourne 1984). This contrasts sharply with a scheme that forces every piece of equipment to fit within a standardized network.
- **Intelligent versus Hard Coded:** The final system should be autonomous and should be able to make its own decisions about the actions it takes under both normal and abnormal conditions (Bourne and Fox 1984). For example, it is not sufficient to have hard-coded responses to fixed situations because there is always some new unanticipated event or situation. A vision system for locating parts is an architectural solution for removing hard constants, and inference rules that can be used to deduce the reason for a machine's failure are a software solution to removing hard coded procedures.

In addition to satisfying these more abstract goals, CML (the control language) must provide an environment in which programming manufacturing systems is drastically easier

Abstract A new computer language for manufacturing is being used to link complex systems of equipment whose components are supplied by multiple vendors. The Cell Management Language (CML) combines computational tools from rule-based data systems, object-oriented languages, and new tools that facilitate language processing. These language tools, combined with rule processing, make it convenient to build new interpreters for interfacing and understanding a range of computer and natural languages; hence, CML is being used primarily to define other languages in an interpretive environment, that is, as a meta-interpreter. For example, in CML it is quite easy to build an interpreter for machine tool languages that can understand and generate new part programs. Once interpreters for different machine and human languages have been constructed, they can be linked together into a system of interpreters. These interpreters can be used to make intelligent decisions for systemwide action planning and diagnostic error recovery. CML is being used in the factory environment to make turbine blade preforms and has proven to greatly simplify the task of building complex control systems.

<TableName>	<FieldName-1>	<FieldName-N>
<EntryName-1>	<DataItem-1-1 or Instruction-1>	. . . <DataItem-1-N>
. . .		
<EntryName-M>	<DataItem-M-1 or-Instruction-M>	. . . <DataItem-M-N>

Figure 1. The Table and Its Parts.

than it is today. To accomplish this, CML provides tools for automatic programming in different target languages (for example, a vision machine language and a robot language) and facilitates the teaching of rules for controlling multiple machine interactions. These tools include a parser driven by a grammatical description (syntax), rule primitives (semantics), and database manipulation primitives (pragmatics). These are the most basic elements for defining, interpreting, and generating text in different languages.

CML also provides a table-oriented environment for both instructions and data. This environment has proven to be a natural way to represent most of the information found in manufacturing. In addition, instructions that manipulate tables are easily “pictured” by people who are not accustomed to the abstractions forced by other languages. The CML table oriented programming environment is not unlike the Lisp environment, which represents everything as lists of lists but which has proven to be quite difficult for non-programmers to learn.

The two-dimensional structures in CML provide direct access to each cell within the structure so that both programs and data can be transformed and generated with traditional database operations. The structure of a table and the names of its parts are shown in figure 1.

Once this choice of primary representation is made, the solutions to many otherwise difficult programming problems are straight forward. For example, a grammar description language can be embedded within a table and used to drive a general parser. In fact, this parser became a standard CML operation.

The table is such a generic data representation that it can easily be viewed in a number of ways. For example, it is both convenient to use as an artificial intelligence (AI) frame and as a relation in a relational database. All that change are the rules of operation; for example, relations are constrained so that no two entries can have the same value for a primary key.

AI languages advertise the fact that they are rule based, frame based, or some other language type. CML attempts to provide the tools necessary to build any one of these language types or some combination of them. In this regard, CML is a bit of a computational chameleon. It gives you the facilities to quickly define a grammar, semantic rules, and data operations so that you can build your own language. This article concentrates on how this language flexibility can be used to solve some of the hardest problems in manufacturing.

Background

CML is designed to integrate systems by using software to overcome the many incompatibilities that are found in the factory environment. Other research groups use different methods to integrate factory systems; the most innovative of which are General Motors Corporation and the National Bureau of Standards (NBS).

General Motors and the Manufacturing Automation Protocol

The manufacturing automation protocol (MAP) is a GM lead multivendor initiative to standardize the communications protocols among the computational components in the manufacturing environment (Adler 1984). The overall scope of the project is to standardize the interconnections of equipment from the level of the communication medium all the way to a complex message-passing scheme. The goal is to allow for file transfers and other application-dependent tasks within a multivendor network. The automation of industry today depends on this kind of standardization effort.

CML enters the manufacturing arena with solutions beyond the scope of the MAP initiative. MAP offers a standardized approach for getting digital information from one place to another, and CML offers the intelligent understanding and generation of these messages. The problem is that machine tools, robots, vision systems, and people all use different languages to express their ideas and actions, and understanding the content of a message often involves making sense of a computer program. CML allows users to easily build a number of “program understanders” and “program generators” that can be integrated into a system to control the whole cell.

NBS and the Advanced Manufacturing Research Facility

The advanced manufacturing research facility (AMRF) project also has the goal of standardizing multivendor communications (Simpson, Hocken, and Albus 1984). This proposed communication model is similar to MAP. The AMRF communications proposal has not been finalized, but it is only a matter of time before a consensus is reached between the MAP and AMRF proposals.

NBS is also developing a general architecture for factory automation. Briefly, it proposes a hierarchical control system with finite-state machines representing the actions of each node in the hierarchy. In addition, there is a systemwide

clock that causes the nodes of the hierarchy to update their state on a regular basis. Even though NBS has developed a strategy for control problems in manufacturing, it has not developed a wide base of generic computational tools.

Although CML is a general-purpose language that can be used to implement the NBS control strategy, it is better used to implement other control strategies. Principally, CML is built to perform transaction-based processing rather than to poll for state changes. This allows the system to be responsive to demanding situations because the central processing unit (CPU) cycles are not wasted on polling and are only used on an as-needed basis. In any event, the actual control is only one small responsibility taken on by CML. The primary emphases of CML solutions involve the automatic planning and generation of part programs.

The History behind CML

In 1981 Carnegie-Mellon University (CMU) and Westinghouse Electric Corporation started working on a large project to automate the production of turbine blade preforms for steam generators (Wright et al. 1982). This system integration problem presented several challenges unlike those encountered in flexible machining systems.

The machines required for the basic forging process were manufactured by companies that were not yet involved in the modernization of controls; an open die forge, a large rotary furnace, and heavy-duty robots with a long arm extension are examples of the machines that had to be reworked. Therefore, most of the effort in the first phase of the project went into the development of the controls at the machine level. In one case, we had to rewrite the controls from scratch. By the time we were ready to integrate the system, most of the allotted project time (through March 1983) had been exhausted. Making matters worse, a 16-bit computer had been selected as the cell host. As a result, a disproportionate amount of our time and effort had to be spent solving memory management problems. Under these circumstances, we were only able to make the cell host into a general node for message passing with rudimentary logical control. The messages that were passed in the system were hard coded, and the cell host had no real understanding of their content.

Forging hot (1200 degrees centigrade) billets into turbine blades, is a hazardous application. Open flames at the furnace door, fast moving robots to avoid excessive cooling of the parts, powerful machinery (that is, hammers in the open die forge striking with 600 tons of force) is adequate reason to keep humans out of the active environment. These hazards are further exasperated by the inevitable machine breakdowns and other unexpected anomalies; hence the need for timely and intelligent machine actions.

At first, we were forced to run the machines in strict sequence so that a human could intervene if necessary. This precaution was a direct result of the host not having enough

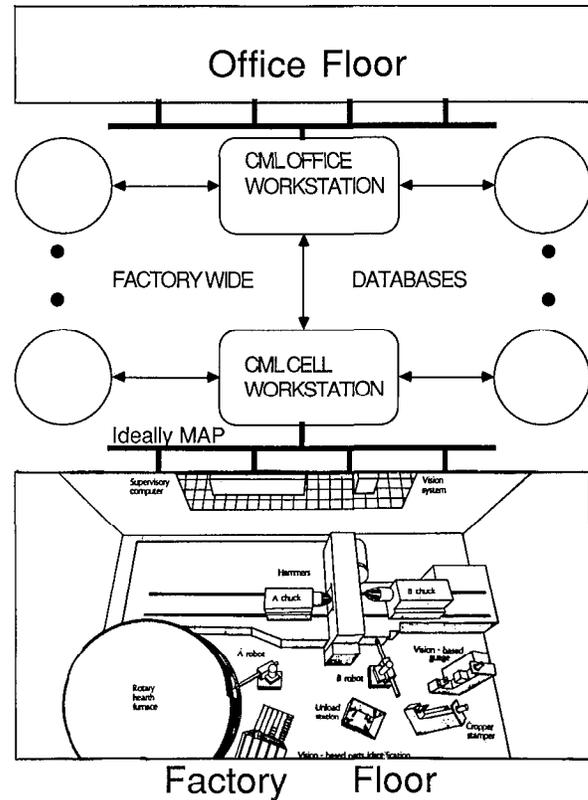


Figure 2. CML's Role in the Manufacturing Environment.

intelligence to understand what was happening in an unforeseen situation. By 1983 everyone involved had agreed that a new approach to the project would be required to satisfy our goal of automating an unmanned cell in a dangerous work environment.

Work began in April 1983 to design a programming language that could both understand multiple languages and communication protocols and provide a simple and extensible approach to system coordination. This language became CML (Wallstein 1985). Figure 2 shows how CML can be positioned in the factory environment to solve many system integration problems. At the bottom of the figure is an early picture of the forging cell (circa 1982). This cell has since grown to include nine machines that work simultaneously whenever possible.

This cell control system in CML performs the following manufacturing system functions:

- Dynamically uploads and downloads part programs
- Automatically builds vision programs based on part definitions
- Dynamically sequences machine operations
- Performs interface and support functions

Figure 2 also shows that CML is going to be used to integrate all factory systems, including those found in the plant office. This article, however, concentrates on using CML to solve the integration problems found on the factory floor.

Machine Specialists

The first step in building a control system for a multivendor manufacturing cell involves developing interpreters, or *machine specialists*, for each device. This requirement is just one of the reasons why it is drastically more difficult to integrate a multivendor system than it is to integrate a single-vendor system. CML provides standard tools for developing these machine specialists, whose job it is to interpret programs and command messages and to generate programs and command messages. Normally, an interpreter is represented by the following relation:

$$\alpha \rightarrow \beta$$

A program in the language α (for example, CML) can be translated into a set of possible actions β (for example, the result of executing a CML program).

CML further augments this idea by allowing the user to set up a number of interpreters within the CML environment. One of these interpreters defines the language α' used by an external device:

$$\alpha \rightarrow \{\alpha' \rightarrow \beta'\}$$

To define a new interpreter, the grammar of this language must be specified as well as a function that selectively maps these programs into actions β' (for example, database operations). Within CML there is one command that binds the grammar and the semantic function into one unit. This command loops over each input sentence; applies the grammar that results in a parse table; and then applies the semantic function to the parser's output, resulting in a final action.

The grammar of α' is defined in a table that is, in turn, input for the parser. Figure 3 shows a grammar that defines simple declarative and interrogative English sentences. The parser starts on the first entry of the grammar (that is, "Sentence") and works its way across the items. The first item in this row directs the parser to the row labeled "Question," where it attempts to find a series of words, separated by spaces, that fit within the appropriate lexical categories defined in figure 4. If this path in the parsing succeeds, then the parser returns to the top entry and usually skips over all of the remaining optional items; this abrupt termination assumes that there is no more input to parse. If the parse fails, then parsing is directed to the entry named "Declare," and the same procedure is followed.

The grammar refers to lexical tables (figure 4) that define the legal words for each syntactic slot. This example does not check for grammatical attributes, such as matching the number of a verb with the number of its corresponding noun. To perform these checks, it is necessary to multiply the number of lexical categories; or to provide an additional level of semantic attachment; or, in this case, to check. We

prefer to use rather weak grammars, with additional checking performed by a set of rules. This approach has the advantage of accepting and making sense out of a wide range of sentences that would otherwise be deemed syntactically invalid.

Each time the parser is given input, it uses the grammar description to help it produce an output table. This parser deviates from most other parsers by producing a linear representation of the parse tree; these table items are essentially the names of the leaves in the tree. In order to keep these leaves unambiguously ordered, special labels (in quotes in figure 3) are attached to the node names to identify the location in the tree. Notice that the entry names in figure 5 are composites of the lexical type and the parse tree's label.

Once the input has been satisfactorily parsed (see figure 5), a function is automatically executed to attach meaning to it. Typically, this function conditionally fires a set of rules based on the input. For example, in figure 6 there are two rules (or pattern-directed *function calls*) that fire only when there is a value in the parse output for each argument in the function call.

After it has been determined that the function "update" should be executed, a substitution table (see figure 7) is built that binds the formal types to their corresponding values. In this way, the function's arguments are also in table format and can be manipulated using the common database mechanisms.

At this point, control would be passed to the function "update." This function replaces the value "baking" in the database table "Parts" by the part's new status "ready," which is found in the substitution table (see figure 8).*

Most machine specialists and their parsers are not designed for input produced by a person but rather are designed to handle intermachine communications. This means that the CML parser must offer a rather unusual set of options to cope with the different technologies offered by multiple vendors. These options include scanning input in different radices, representations, byte orders, and other data configurations.

We have used this tool to understand programs that control robots, vision systems, machine tools, and people. We have also found that it is a useful approach to building new tools. For example, the CML assembler is a CML program that converts a text representation of CML into its internal representation. This would have been a major project in other languages but was a one-day project in CML.

Program Generation

An interpreter is usually expected to "understand" programs and then to take some appropriate action based on that

* Notice that the address of an item is represented as the triplet [tablename,entryname,fieldname] separated by colons and that numeric indices can be used optionally instead of names.

Grammar	Field1	Field2	Field3	Field4
Sentence	Grammar:Question	Grammar:Declare		
Question	verb/label="-quest"	art	noun/label="-subj"	adj
Declare	art/label="-declare"	noun/label="-subj"	verb	adj

Figure 3. Grammar for simple declarative and interrogative sentences.

verb	number	robots	number	adj	art
is	single	parts	plural	ready	the
are	plural	part	single	baking	a
were	plural	robot	single	titanium	all
was	single	robots	plural	steel	each

Figure 4. Lexical Tables Named in the Grammar Description.

understanding. However, in a limited way, the interpreters described in the last section must also be able to “speak” the language. This usually involves three additions to an interpreter, making it a complete machine specialist:

1. Specification Language: This language is the internal representation for specifying an external machine’s problems.
2. Algorithm: This algorithm converts the problem specification into the appropriate program structure.
3. Grammar: This grammar describes the surface representation of the message that will finally be sent to the external device.

Conveniently, these three additions are the same mechanisms that are used for program understanding; however, they are now used to generate programs instead. It is usually difficult, though, to make the input and output language mechanisms identical.

CML facilitates automatic program generation in several key ways. Because all programs and specifications are represented in the database, new programs can be built using inter-table operations and modified using intra-table operations. The inter-table operations provide a convenient abstraction of an otherwise detailed program so that a simple formula can be used to describe how the program segments should be pieced together. An example of such a formula is:

program ::= *header* + *repeating-body* + *trailer*

This formula, together with a part description, can be used to generate an algorithm to either machine or inspect the part.

In this case, the program is accumulated by pasting the *header* onto the table that represents the final program. Then, given a skeletal version of the *repeating body*, the intra-table operations can be used to update the key values. These updated program segments can be accumulated on the final program within a control loop. Finally, the trailer can be appended, completing the program. The constructive approach illustrated by this formula has been successfully used

Input > *is the part titanium*
 Answer > *Yes*
 Input > *the part is ready*

\$Parse	Value
art-declare	the
noun-subj	part
verb	is
adj	ready

Figure 5. Parser Input and its Output.

to generate part programs for machine centers and inspection devices. As a result, these programs no longer have to be written by hand for every part style.

Once a program has been built, it usually has to be transformed into a final surface structure that is acceptable to the external machine. For example, the MCL program shown in figure 9 has to be compacted, blocked into fixed block size, and preceded by a partial program command (notated “C,TPP”).*

This final “grammatical” transformation is accomplished by two commands. The first step compacts the entire table (figure 9 into one string), and the second command blocks the strings and attaches the required syntactic markers. The result is shown in figure 10; and this is sent entry by entry to the receiving machine controller.

Now that we have developed a machine specialist which can both understand and generate programs, it is possible to inter-translate between multiple languages. This is an important feature in manufacturing because even when machines use the same control language, the machines themselves are usually different enough to require minor translations. For example, two virtually identical machine tools might have their axes labeled differently, and cutter offsets almost always vary from machine to machine. This is the same problem that two English speakers can have. For example, generational and geographical differences can change the language enough to require translation.

System of Interpreters

Once machine specialists have been built for each machine, their performances must be orchestrated into a coherent sys-

* Machine control language (MCL) is a standardized language for controlling paper tape driven numerically controlled (NC) and computer driven numerically controlled (CNC) machine tools. It is slowly being replaced by higher-level languages because of its lack of control structures.

Rules	Eval	Func	Arg1	Val1	Arg2	Val2	Arg3	Val3
R1	eval	lookup	verb-quest	x	noun-subj	y	adj	z
R2	eval	update	art-declare	x	noun-subj	y	adj	z

Figure 6. Simple Rule Set for Semantic Attachment.

\$Sub	Value
art-declare	the
noun-subj	part
adj	ready

Figure 7 Substitution Table Produced by Rule Execution.

Update	Command	Argument1	Argument2
c1	update	\$sub:adj:value	parts:order1:status

Parts	Status	Material	Batch
Order1	baking	titanium	101
Order2	ready	steel	93

Figure 8. Update Function and Simple Database for Part Descriptions.

tem. This system must be responsive to the external environment, and it must be cautious about its choice of action.

Each machine specialist is responsible for synthesizing complex messages into atomic units (*state names*). These atomic units then become the lexical names for a new language: one for the whole cell. In this new language, a process plan can be written for each machine in the cell. In order to accomplish this, however, each logical state must be decomposed into three substates:

- state-R: The machine is “ready” to execute an action.
- state-W: The machine was asked to execute an action, and we are “waiting.”
- state-C: The machine told us that it “completed” the execution of an action.

Each machine has its own process plan, and as this process plan is executed the current state of each machine is recorded in a global table of Cell-States (see figure 11).

The messages from single machines are parsed and a set of rules is “expanded” to assign and execute the message’s meaning. In fact, the output of the parser and the cell-state table can be represented in isomorphic structures (see figure 12). The entry names of the tables are the general categories: syntactic categories and machine-state categories, and the first item in each entry must be a valid member of its category. The legal values of these categories can be any value that properly belongs as its member. Once a table is in this format, it can then be used as the data set of information that is matched with a rule set.

The expand command, shown in figure 13 associates a data set with a rule set and then “fires” the appropriate rules.

Figure 14 illustrates the internal arrangement of the software making up the system of interpreters. In a typical CML system, there are several active processes: CML itself and processes that manage data from external lines at an interrupt level. In addition to these processes, others can be added that perform application-dependent functions. For example, an emulation package that allows us to replace any number of real machines with a program which returns the expected machine tool response was implemented.

Figure 14 schematically shows interprocess communication by way of mailboxes. These mailboxes are located in shared memory and are supported by background procedures that provide features similar to those expected of a human-oriented mail system, for example, sending a message to a named process, forwarding, making carbon copies, and logging.

A message (M1) enters the system at interrupt level and is placed directly in a CML device mailbox. After the message is completely received, it is automatically transferred (becoming M2) to the protocol’s agenda box. In this way, the procedure can read all of its mail from one point in the program; otherwise, the procedure could easily get hung up waiting for mail from one source while it is receiving new mail from another. The protocol handler strips off the protocol information and decides whether the message is valid. If it is not, the handler informs the sender; otherwise, it forwards the data portion (M3) of the packet to the CML agenda mailbox.

At this point, all of the data processing is done within a CML application program. The application program reads the next task on its agenda and determines how it should be processed by looking at routing information in the “system dispatch” workspace.*

In the dispatch area, there is a table called “\$Lang” that provides appropriate information, including the workspace which specializes in this kind of message, the name of the grammar table to parse it, and the function to evaluate after parsing is completed. The message (M3) then pursues this course (see Figure 15); getting parsed and processed by the

* CML provides a mechanism for grouping functions into private workspaces, similar to IBM’s programming language (APL) workspaces. However, in CML there are formal mechanisms for interworkspace communications, so that in fact the workspaces act more like objects in SMALLTALK. At the same time, these workspaces give structure to a large program similar to that given by modules in structured languages.

Program	Block	A1	A2	A3	A4	
e1	N1	G00	G90	B-31.		<i>Header</i>
e2	N2	Q03				<i>Body-1</i>
e3	N3	G90	X2.75	Y23.	B-45.	
e4	N4	Q03				<i>Body-2</i>
e5	N5	G90	X2.75	Y27.	B-45.	
e6	N6	M2				<i>Trailer</i>
e7	N7	G00	G90	Y30.	B0	
e8	N8	M30	M40			

Figure 9 Final Program Structure.

\$Parse	f1	f2
e1	C,TPP	\$N1G00G90B-31.*N2Q03*
e2	C,TPP	N3G90X2.75Y23.B-45.*N
e3	C,TPP	4Q03*N5G90X2.75Y27.B-
e4	C,TPP	45.*N6M2*N7G00G90Y30.
e5	C,TPP	B0*N8M30M40

Figure 10. Final Surface Structure

Robot-Process-Plan	Cell-States	Value
Load-Furnace-R	Robot	Load-Furnace-R
Load-Furnace-W	Furnace	Open-Door-C
Load-Furnace-C	Tool	Machining-W
Load-Tool-R		
Load-Tool-W		
Load-Tool-C		

Figure 11. Process Plan and Cell-State Summary.

appropriate specialist. Finally, the specialist sends back its new state information (M4) to the "cell manager."

As a result, new rules are fired, and new command messages (M5) are generated for the next step in the cell's operation. These messages also are directed through the system dispatcher and the final result is an action taken by a machine specialist. The machine specialist builds a command suitable for an external device and sends it out (M6). Finally, the protocol handler encodes it for transmission and sends it over the serial line for execution. In this example, the message (M1) could have come from a vision system providing coordinate information destined for the robot. The intervening messages are internal status messages and intermediate and final translations.

This array of activity utilizes three principal ideas so that the flexibility of the resulting control system is maximized:

- **Events:** CML is event driven. Commands and messages can be sent at any time without fear of them being lost or mishandled. They are processed in a priority-ordered first-in, first-out scheme.
- **Agenda System:** CML maintains a list of its activities and it can make decisions about when and where these activities should be processed. For example, in critical periods it is important to not start a job that requires too much effort; during a manufacturing run, it is wise not to start preventive maintenance.
- **Rules of Interaction:** The cell specialist is primarily concerned with the interaction of the machines. When a

message is received from a device specialist, the state of the specialist is updated, and the rules are scanned.

This software architecture has proven to be very robust and seems to be flexible enough to handle all sorts of situations. The next section concentrates on how a system of this sort can be built with a minimum amount of effort in CML.

Teaching Cooperative Actions

Teaching sequences of fixed actions to robots has been standard fare since the robot revolution of the late 1970s. However, this approach of "teaching by doing" has never been successfully extended to complex conditional sequences, that is, programs which are necessary to represent complete robot programs, and multiple machine interactions. The solution to this problem is apparent within the context of a CML environment.

Robots cannot move inside a furnace without opening the door. Furnace doors must be closed as robots depart, or hydraulic hoses will melt. To avert potential catastrophes such as these, the machines must be carefully interlocked. This interlocking is accomplished in CML by using rules that limit the conditions under which actions can occur.

There are several approaches to the generation of new rules. The first approach is the obvious one: think of and write all the necessary conditions for each action. Although this approach is tedious and often quite difficult, it does have two advantages: (1) the rules can be concisely constructed to be logically minimal and (2) the rules are not temporally ordered and fire whenever an action is possible. Unfortunately, manufacturing cells are often assembled by nonpro-

\$Parse	Value	Cell-States	Value
Art-1	The	Robot	Loaded-Part-C
Noun-1	catalog	Furnace	Door-Open-C
Verb	has	Tool	Machining-W
Adj-2	four	Vision	Finding-Part-R
Noun-2	programs		

Figure 12. Comparison between Message Parse Table and Cell-State Table.

grammers who are not accustomed to building logic-based assertions. To avoid catastrophic errors, CML provides a second, automatic approach to the development of these rules.

This second approach allows the rules to be “taught,” using simple graphics and pointing, for example, by using “mouse” technology. Each machine is represented by an icon that can be referred to by a pointing action. When there is a place in execution where no rules are found to execute, then a person is given the chance to add a new rule for the situation. The rule is constructed from the current system’s state (see figure 11) and can be paraphrased as follows:

*if the-cell-is-in-this-state
then proceed-with-indicated-action*

When written out as a pattern-directed function call (see figure 16), it is apparent that the rule is just a simple transformation of the state table (see figure 12). This rule is then automatically added to the rule set. This rule set is matched to the cell-state table, and if no rules are fired, then the otherwise clause is executed (Build-NewRule in figure 17). However, because a rule was just added to explicitly match the current state configuration, it fires. This function results in the machine advancing to its next state, and the rules are checked again. This time if a rule is found, the cell continues execution, and if not, the operator points, a new rule is built, and execution continues.

If each state of the cell were equally likely, then this approach would not be very effective because the number of rules would explode exponentially. Fortunately, this simple algorithm generates the key rules after only a few cycles. In our experience, it took about 4 cycles in a cell making 1 part style with 9 machines and 1 bottleneck. If the number of part styles is increased, then the timing variations of each manufacturing step start to drastically increase the number of situations that haven’t been previously encountered, thus increasing the number of necessary rules.

Because it is our goal to use this approach in small machine cells as well as in large flexible systems, it is useful to automatically reduce and sometimes weaken the generated rules. The process of rule reduction turns out to be simple because it usually involves striking conditions from rules when the states in a machine’s process plan have been covered. For example, suppose that a robot can be in two possi-

ble states (*is-working, is-waiting*), and suppose that there are the following two rules:

*if the-robot is-working then start-gage,
if the robot is-waiting then start-gage*

These two rules can be reduced to one rule without conditions. Michalski (1980) reviews this inference rule and four others that effectively reduce the rule set, and Hayes-Roth and McDermott (1978) suggest the similar approach of rule abstraction. This example presents the weakest form of induction and amounts to nothing more than logical equivalence. However, there are other less stringent ways of reducing rule sets. For example, meta-rules can be added to the system that determine when rule conditions can be stricken, thus weakening the rules and broadening the conditions under which they apply. We experimented with a number of these meta-rules. The simplest one removes conditions and reduces rules when the majority of a state domain has been covered. For example, if we extend the state domain for the robot to three states (*is-waiting, is-working, is-idle*), then this meta-rule allows the same inference we already made from the given rule set, thus assuming that it is safe to gage when the robot is idle. Unfortunately, this rule can be quite risky because the inferred rules cannot be considered safe for critical machine interactions.

There are safer heuristics for inductive inference. In physical systems, such as in manufacturing, there are physical spheres of influence; in other words, machines that are in reach of one another must collectively be programmed more conservatively than machines which are geographically far apart. This forms a rather natural two-tier rule system, where the first layer of rules considers the geographical reachability between machines and only then does the second layer of rules consider the relative machine behaviors.

Teaching these reachability rules can also be accomplished within a graphics-oriented system. A simple approach would be to draw concentric circles around a machine until the scope of each machine’s influence is circumscribed. This data also provides information about other aspects of the system; for example, it can be deduced that machines with a broad reach are probably being used for material transport.

We are continuing research in this area, extending these ideas to larger flexible manufacturing and assembly systems. In order to avoid the exponential rule explosion, these larger systems demand heuristics for selecting and combining rules. Fortunately, many of the required heuristics are not needed for machine interactions but are needed for planning factory schedules.

Installation of AI in the Factory

It is difficult to install any advanced technology in the factory today unless there is at least one trained specialist. These advanced technologies pose problems just because they are advanced and rarely, if ever, can be purchased as turnkey

MAIN > expand, cell-states, rules

Rules	Eval	Func	Arg1	Val1	Arg2	Val2
R1	eval	act	Furnace	=Door-Open-C	Robot	=Loaded-Part-C
R2	eval	act	Robot	=Loaded-Part-W	Furnace	=Door-Open-C

Figure 13. Rules of interaction.

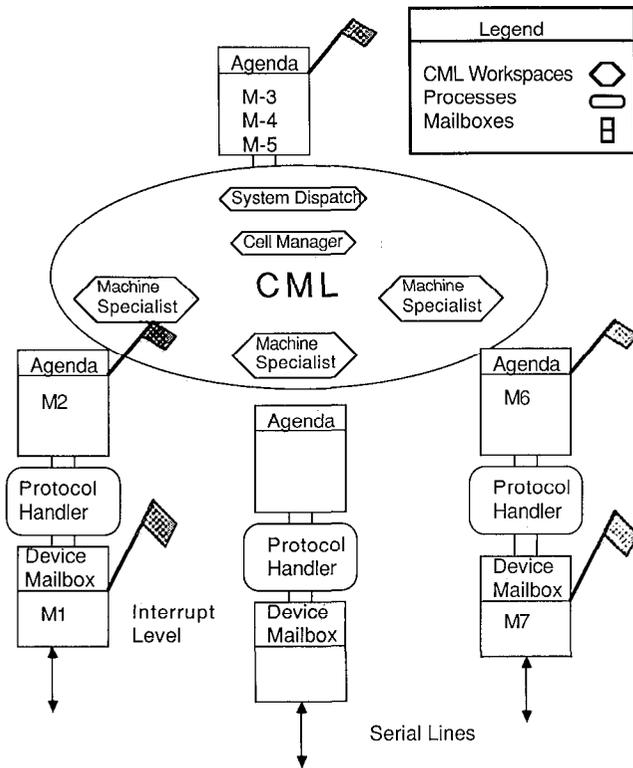


Figure 14. Data Flow in a CML System.

systems. This has led us to introduce to the factory AI tools that are designed to increase the productivity of system builders. The long-term goal is to make it possible for non-programmers to manage large software projects.

Figure 18 illustrates an array of these tools that have made it possible for nonprogrammers to develop the control system for a complex manufacturing cell. The "y axis" of figure 18 describes how the CML tools are used. The automatic tools can "generate" and "understand" programs that fit within a defined family (the definition of this program family still has to be produced "by hand"). On the other end of the y axis are tools that aid the engineer in the remaining programming tasks. First, some of the programs can be "taught" instead of programmed. Second, a special tool has been built that is itself an expert in CML.

The "CML expert" takes a CML program as input and then writes a letter to the programmer. This letter gives rec-

\$Lang	Parser	Evaluator	Workspace
Robot	VAL	Interpret	Robot-1
Vision	RAIL	Interpret	Vision-1
Furnace	Ladder	Interpret	Furnace-1

Figure 15 Some of the information in the system dispatch workspace

ommendations for program optimization, comments on program style, warnings about obsolete functions, and advice about program structure. This kind of tool is important in the manufacturing environment, because it cannot be assumed that the users are proficient in programming.

Figure 19 (see page 96) is an example of a letter written by the CML expert. This letter is made from a series of pre-programmed paragraphs. These paragraphs are invoked using a rule-based system in CML and are accumulated in an outline. Once the first selection of paragraphs is made, the outline is scanned and modified so that redundant paragraphs are removed and appropriate concluding paragraphs added. Hopefully, the final letter can then be used by the programmer as a guide for improving the program. This simple approach to letter generation is being extended to independent sentence and sub-sentence generation so that the criticisms of a CML program can be uniquely tailored to pinpoint its most basic problems. In addition, the very strong similarity between this approach to letter writing, and the program generation in figures 9 and 10 should be noted.

Current Status

The first CML application was in production for several months during the fall of 1984. During the winter of 1985, the two robots in the cell were replaced with two new robots and two new controllers. The new CML software was written and tested before the cement that glued the robots to the floor had dried.*

Recently, we added two more machines to the cell, and only a handful of additions to the CML control program were necessary. These new machines used a controller which was

* Messages that are normally sent to machines can also be sent to an emulation package, which sends back a meaningful response in a timely way. This allows us to test the control program before the actual machines are ready for testing

Rule	eval	func	F1	V1	F2	V2	F3	V4
R1	eval	act	Robot	=Load-Furnace-R	Furnace	=Open-Door-C	Tool	=Machining-W

Figure 16. New rule constructed from current cell state

Eval-Loop	Command	Arg1	Arg2	Arg3
e1	while	~ExitLoop		
e2	expand	Cell-States	Rules	Build-NewRule
e3	end			

Figure 17. Top level evaluation loop.

the same as another already in the system, and all that was necessary was to define the process plan for the machines and the name of their external serial-line part.

CML is now being used as the basis for several new projects. Some of the projects are being done at CMU and others are beginning at Westinghouse. One project at CMU, called the machinist expert, has the goal of capturing the "sensory" knowledge—in this case, the "feel" that is required to make a part. Machining parts accurately and without error will involve automatic tool and fixture selection, machine path planning, and in-process inspection. Most importantly, though, the expert will have to make the decisions that are necessary when there are machine failures, programming errors, and tolerance errors in parts. These problems require intricate strategies for collecting and reasoning about sensory information.

CML continues to develop as it meets new challenges in AI and manufacturing. Already it has made the integration of multivendor systems significantly easier, and Westinghouse is commercializing it.

Summary

CML provides a group of computational tools that facilitate building a system of interpreters, automatically programming solutions to manufacturing problems, and coping with the communications problems that are prevalent in manufacturing systems.

We set the goal of integrating new factory systems without using any traditional programming that is, typewritten logic. This remains an ideal, but we have successfully realized a significant reduction in the programming time for these systems. The programming that remains will be further reduced as we accumulate a large library of device interpreters. Much of the traditional programming is alleviated by a convenient teaching environment, written in CML, and the ability to automatically reprogram new solutions for controlling irregular situations.

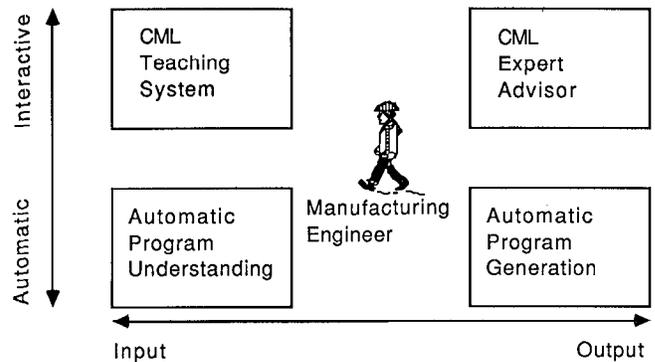


Figure 18. AI Tools for Increasing Software Productivity.

Acknowledgments

This project has only been possible because of generous grants and support from the Westinghouse Electric Corporation. At Westinghouse, I would especially like to thank Jerry Colyer for his long-term efforts. I would also like to thank Mark Fox for the opportunity to write this paper and Richard Wallstein for his editing advice.

References

- Adler, M B 1984 GM Manufacturing Automation Protocol In CAM Symposium, Univ of Cincinnati, 159-170
- Bourne, D A., and Fox M S 1984 Autonomous Manufacturing: Automating the Job Shop. *Computer* 59(243):76-88.
- Fussel, P ; Wright, P K ; and Bourne, D A 1984 A Design of a Controller as a Component of a Robotic Manufacturing System *Journal of Manufacturing Systems* 3(1):1-11.
- Hayes-Roth, F., and McDermott, J. 1978. An Interference Matching Technique for Inducing Abstractions *Communications of the ACM* 21(5):401-410.
- Michalski, R. S 1980 Pattern Recognition as a Rule-Guided Inductive Inference *IEEE Pattern Analysis and Machine Intelligence* (4):349-361
- Simpson, J A ; Hocken, R K.; and Albus, J. S. 1984. The Automated Manufacturing Research Facility of the National Bureau of Standards. *Journal of Manufacturing Systems* 1(1):17-32
- Wallstein, R S., ed. 1985. *CML Reference Guide*, The Robotics Institute, Carnegie-Mellon Univ
- Wright, P K.; Bourne, D A.; Colyer, J. C ; Schatz, G C ; and Isasi, J 1982 A Flexible Manufacturing Cell for Swaging *Mechanical Engineering* 104(10):76-83.

Jun 10, 1985

Dear David:

I have spent some time analyzing the functions you have used in CML, and have come to a number of conclusions. Nothing I say here should be taken as absolute, since the analysis is only performed at a surface level.

A huge portion of your program is devoted to input and output operations. Perhaps you should try to centralize some of these activities in one module.

You have used quite a few of the control features, which indicates to me that you have spent quite a bit of time refining the program.

You have used a number of database oriented operations, which tells me that you have appreciated at least this aspect of CML. . . .

I have noticed that you tend to use quite a few begin conditions. It is usually quite easy to bring these together into a set of rules, that will show the logical structure of a program at a glance. The other advantages of using rules is that they tend to run faster than their begin counterparts, it is often easier to change the program to manage more situations, and to include parameters that allows the overall function to be much smaller.

Input statements are often convenient to use if you are writing a testprogram, but they should not be part of any production software. The problem with using inputs is that they do not verify that the input-string is in fact in the range of appropriate responses. Try to use the connect statement instead.

I noticed that you are using a printcommand. These are often convenient in test programs, just as a quick way to write out some sort of message. However, you should seriously consider switching these to generate commands, since the output of a generatecommand can be easily redirected to external processes and files.

Thank you for trying the CML expert system, I hope my comments about your program have proven to be helpful.

Sincerely,

The CML Wizard

Figure 19. Letter Automatically Written by CML Expert.