A White Paper

State Language for Machine Control

by Kenneth C. Crater, Chairman
Control Technology Corporation

Introduction

State languages have recently come to the forefront of discussions on superior approaches to automation programming. The temptation is to view state language as a "new" automation tool, since it has only recently received attention in our industry's trade journals. In fact, the use of state languages in control dates back at least to the late 1970s, when the early implementations of Quickstep[1][2] were developed to mimic the function of electromechanical cam programmers. The theoretical basis of state language control is older still, having its foundation in Petri Net theory as developed and described by Carl Adam Petri[3] his PhD thesis in the early 1960s.

There is still much confusion as to what constitutes a state language, how best to use one for automation applications, and what economies exist in their adoption. I hope to clarify some of these questions in this paper. Further, I will show that the increased use of state language technology in automation is a direct response to some very specific new demands being placed on manufacturing organizations. Far from being "just the next new language", state languages came into existence to address problems which could no longer be adequately addressed using preexisting programming techniques.

Why State Language? Why Now?

The emergence of state language as a preferred programming paradigm is not occurring in isolation. Rather, it reflects sweeping changes taking place in the practice of industrial automation. Referring to Figure 1, each stage in the development of automation technology was both a reflection and a determinant of the underlying technologies and techniques used by practitioners.

In the early days of hard automation, stand-alone fixed-function systems were the predominant practice. Information was transported via clipboard (the physical kind), actuation was largely mechanical (transitioning to pneumatic/hydraulic), and programming and user controls were hardwired. These practices were self-consistent, and were also consistent with the primary goal of automation at that time: to reduce labor input and decrease unit cost of production.

	early automation →		*present day*
Automation Level	**Hard Automation**	**Flexible Automation**	**Intelligent Automation**
System Integration	**Isolated Systems**	**Interfaced Subsystems**	**Full Integration**
Information Realm	**Clipboard**	**Workcell**	**Intranet**
Motive Technology	**Mechanical**	**Pneumatic/Hydraulic**	**Servo/Stepping Motors**
Programming	**Hardwired Logic**	**Relay Ladder Logic**	**State Language**
User Interface	**Control Panels**	**MMIs**	**Website**

Figure 1. The evolution of automation has, by necessity, been accompanied at each step by parallel changes in all underlying tools and technologies.

The quickening pace of global markets, commencing in the 1970s and continuing through the 1980s, drove the need for flexible automation. Among the new technologies employed to meet this challenge were PLCs, increasingly integrated with minicomputers and later with personal computers, organized in workcells within which configuration and production information was shared. It was in this environment that the innovation of relay ladder logic thrived, allowing a modest degree of complexity in automation strategies and providing the decision-making tools for flexible machines. This language also allowed a further degree of flexible through reprogramming, reducing the cost of responding to changing market needs.

Again, this was an internally-consistent toolset, adopted in response to environmental conditions. Flexible tooling allowed the increasing cost of automation, and floorspace, to be shared among several product variants, and improved the ability of production management to respond to quixotic markets.

Now in the mid-1990s, automation techniques are being driven by *information* requirements, coupled with an unprecedented need for agility. The factors in this environment which directly affect the selection of automation technologies include:

Dramatic market and technological change – Agility has been called *that characteristic which allows an organization to thrive in an environment of constant and unpredictable change*[4]. In the face of such change, companies are adopting strategies which provide the greatest ease of retooling, and the greatest extent of flexibility without retooling. Among these strategies are the use of more highly integrated systems and corresponding languages that reduce the costs of interfacing motion control, data acquisition, communications and other requisite technologies. This paper will show how state language can accommodate this higher level of integration and thus decrease development time substantially.

Broadened expectations of quality – The definition of quality has changed, and it has become a non-negotiable expectation in many markets. Two critical technologies for implementing quality programs are analog data acquisition, for the purpose of making qualitative measurements, and communications for presenting data to remote monitoring, storage and retrieval systems. Previous control techniques were ill-equipped to accommodate these requirements, and often required supplementing with secondary systems. State language, however, provides a mechanism for integrating these requirements with the control program in a concise and meaningful manner.

The advent of the information age – Every major manufacturing company has in place an extensive information network, and either has or will extend that network to the plant floor. The same expectations of access, visibility, and control which have come about in the office environment are now making their way to the plant floor. Older, boolean-based programming methods for automation do not present an information-rich resource, however, and fail to meet these expectations.

The shifting role of the manufacturing worker – Initial automation efforts were focused on extending the physical capabilities of human workers—exerting more power, extending their precision, allowing work in intolerable environments. Then, the focus moved to cost savings, increasing output per labor hour expended by replacing human effort with machine effort. In the process, however, the intellectual contribution of the worker was lost—the inflexible machine was uninterested in the human

operator's opinion of the quality of output, therefore the worker lost the ability to make a difference. The inevitable result was a demotivation and de-skilling of the workforce.

Now, however, many companies are recognizing the necessity of recapturing the value added by the human workers' intellect. Ironically, this sometimes points to a conscious limiting of the extent of automation, providing some measure of additional qualitative control which may be exerted by the operator[5]. Maintaining reasonably high levels of automation, while simultaneously allowing such qualitative control requires automation systems with additional capabilities. Mathematical functions, extensive parameter storage and communication, enhanced operator interface facilities, sensing / reporting capabilities and integrated motion control for making mechanical adjustments are all enabling technologies to meet this requirement. The simple control languages of the last decade fail in many respects to fulfill these needs, although it will be shown that the state language framework can do so.

The Framework of State Languages

The appeal of the state language framework resides in its simplicity and its fit with the problems being addressed. In the discrete manufacturing world and, to some extent, in the batch and continuous process worlds, these problems often consist of a series of steps or states which a machine must go through to perform a series of operations on a workpiece or product batch. Ignoring, for a moment, the more complex case of asynchronous operations, a state language can exactly mimic the structure of this type of problem.

The fundamental tool of a state language is the state, or "step". A step defines the complete status of the machine or process (or a portion thereof) for a finite period of time. Typically, this status consists of two components:

a. One or more commands to create a motion or change, thus causing a new physical state to be adopted by the machine or process.

a. One or more instructions to limit the duration of the step, and specify the next step to proceed to upon completion of the current step.

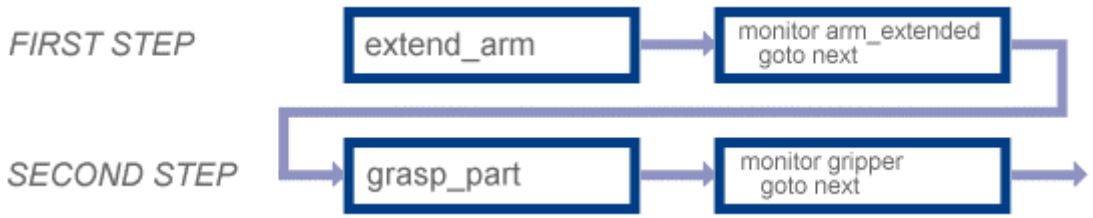

Figure 2. The components of a simple state language program, showing motion commands (left) and instructions for proceeding to a new state (right).

Note that either of these components may be arbitrarily complex. The motion command may be as simple as a single digital output change, perhaps to actuate a pneumatic solenoid valve. However, it may instead be as sophisticated as multiple commands to initialize and turn a series of servo motors, specifying motion and tuning parameters as well as terminal positions for each.

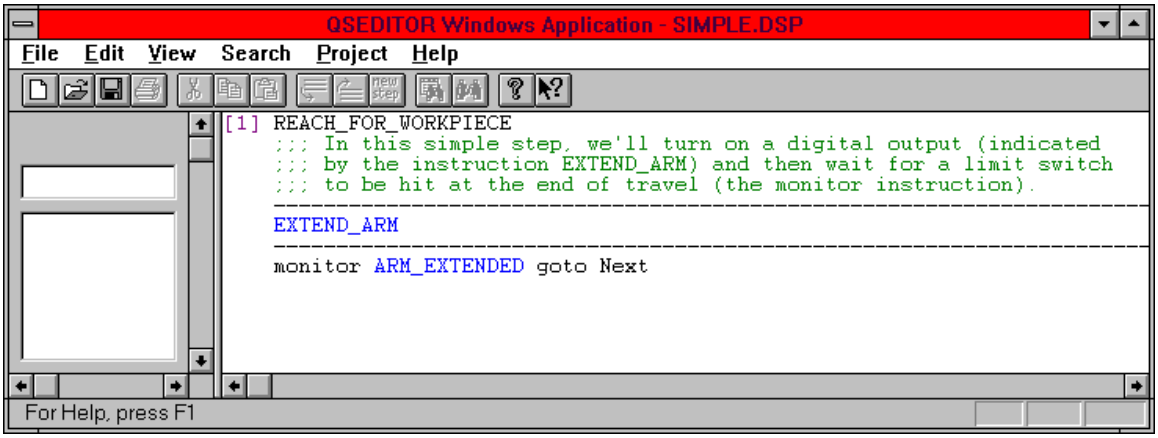

Figure 3. A simple step implemented in a practical state language, Quickstep.

The instructions for terminating a step may be as simple as a time delay instruction, or an instruction to monitor a single digital input – perhaps wired to a limit switch. However, they may get as complex as instructions to read multiple analog input values, representing temperatures, positions, and other variables, combine them mathematically, and decide upon a new direction for the program to take depending on the outcome.

The flexibility provided by this framework is quite powerful. Because the framework allows very high-level commands to be implemented, a state language can "keep current" with evolving actuation and sensing technologies. But the real advantage may best be seen by examining a state language program in overview.

The structure of the resulting program bears a striking resemblance to a flowchart of the desired operation of the machine. This one-to-one correspondence is responsible for much of the savings in development, debug, and maintenance time attributable to state language use. It provides a further benefit by demystifying the operation of a control program, allowing other members of a design team to understand, review, and comment upon the program's structure. The program listing may therefore become a common point of communication among the controls engineer, machine designer, process engineer or industrial engineer, information systems engineering, maintenance personnel, and even the machine's operator. This creates an environment where each team member is able to maximize their contribution to the automation effort.

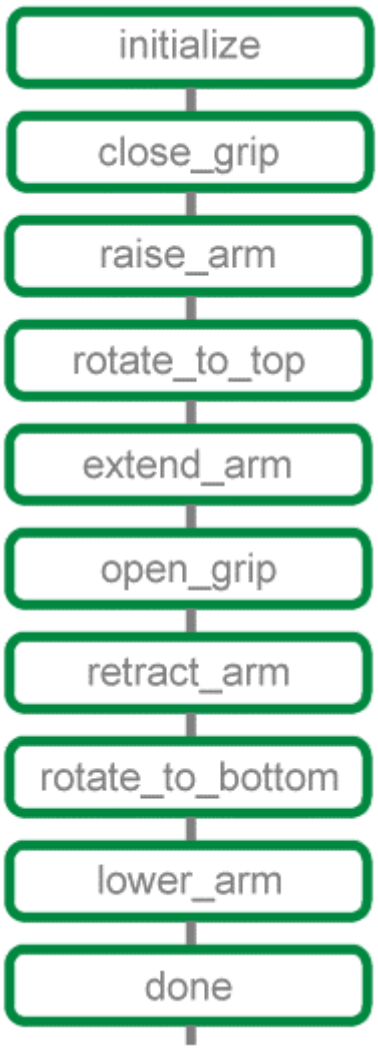

Figure 4. The structure of a simple state language program.

A Comparison to Relay Logic

By far the most prevalent language in the control of automated machinery today is relay ladder logic (RLL). The purpose of RLL was to provide a language to emulate the functionality of the electromechanical relays

which had previously been used to build logic systems for control. Before engaging in a comparison, it must be noted that the adoption of RLL was an essential step in achieving the acceptance of electronic controls in the factory environment. The use of this language allowed the plant electricians who maintained the previous electromechanical controls to become familiar with a new generation of electronic controls. At the same time, RLL facilitated the migration of old control schemes from electromechanical methods to electronic methods.

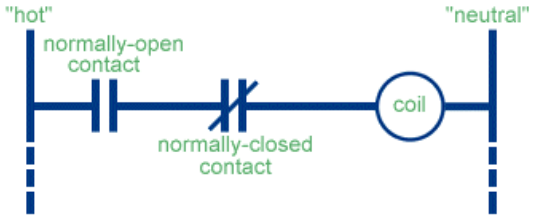

Figure 5. The basic elements of Relay Ladder Logic.

The essential elements of RLL are the *coil*, analogous to the electromagnetic coil of a mechanical relay, and the *contact*, typically associated via labeling with one of the coils which "actuates" the contact. These elements may be seen in Figure 5. Contacts may be of one of two types: *normally-open* or *normally-closed*. Normally-open contacts are said not to pass any "current" when the associated coil is inactive. Conversely, normally-closed contacts pass current only when the associated coil is inactive. In either case, the current, of course, is imaginary and is used only to represent the behavior of the program.

In actual practice, the contacts of an RLL program are used in various combinations, in such a way that they model a Boolean equation. An example of such a combination, along with the equivalent equation, is shown in Figure 6. When contacts are paralleled, their operation is the equivalent of a Boolean OR function. When contacts are in series, their equivalent function is the Boolean AND.

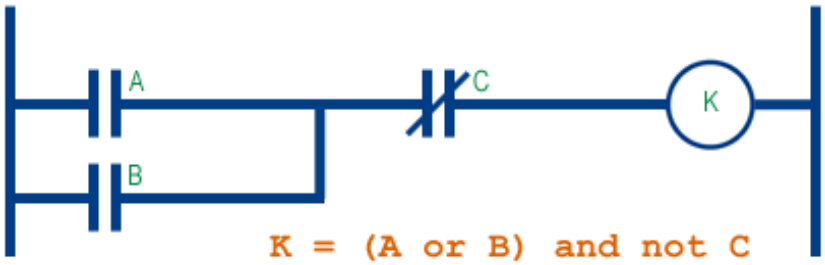

Figure 6. A typical rung and the equivalent Boolean equation.

The runtime execution of such a diagram consists of a cyclical scanning of the entire diagram, examining the current state of all contacts and then determining which coils should be active at that moment. The contacts associated with the active coils will, on the next scan, assume their active state which may in turn affect the status of other coils, etc.

In practical machine control applications, the use of RLL brings with it substantial additional overhead. Most automated machines have a natural sequence of events by which they convert a raw material or unfinished workpiece into a finished product. This sequence typically consists of a series of mechanical states the machine must assume, driven by the control system. To program such a series of states using RLL, it is necessary to use a number of latches, with each latch being built from a number of RLL elements.

Figure 7 illustrates this technique. An external event, sensed by *LIMIT_SWITCH*, provides energization to the coil *M1*. The contacts from *M1* then close, bypassing *LIMIT_SWITCH* and providing continuous "current" to the coil for *M1*. This coil then remains latched, even if *LIMIT_SWITCH* de-energizes. Additional contacts from *M1* may then be used elsewhere in the program to enable those events which should only occur subsequent to this portion of the machine's sequence. This is illustrated in the second rung of Figure 7, where a contact from *M1* in conjunction with a second external input, *LIMIT_SW2*, may then engage another coil, labeled *M2*. In this case, a normally-closed contact from *M2* then opens and de-energizes the coil *M1*, signaling a progression from one state to another.

The necessity to build "statefulness" into a program through the programming of latches may not be a significant inconvenience in a small program, but as programs grow in complexity the additional burden added by this requirement becomes substantial and cumbersome.

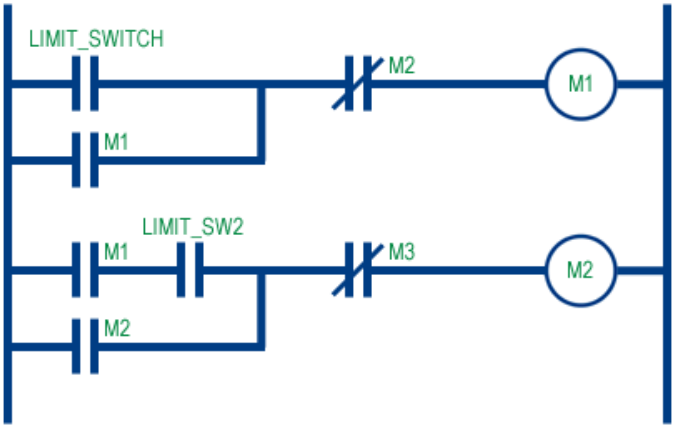

Figure 7. Establishing statefulness using Boolean latches.

The proliferation of many non-Boolean technologies in automation has also greatly diminished the efficiency of RLL as a programming language. Servo and stepping motor control, analog data acquisition and control, and intensive data gathering and communications were not predicted by RLL, nor are they adequately accommodated. Such control requirements are usually addressed in RLL by injecting the new element of a *function block*, which occupies the place of a coil in the relay diagram. The function block is triggered by one or more contacts, and then executes whatever instructions are contained within it. Often, these are expressed as a series of codes to be sent to a dissimilar motion controller, communications port, etc.

The necessity of supplementing RLL with instruction elements of a completely different paradigm, coupled with the lack of a pre-existing structure for the most common requirement of machine control, has now become a substantial obstacle to contemporary automation efforts. In large part, the increasing acceptance of state language technology is a response to that fact.

State Language Contrasted With SFC

The field of programming languages for automation has seen much activity in recent years, again as a reflection of changing needs. One of the more visible efforts has been the move to standardize a group of older languages, mostly Boolean or procedural languages, under the nomenclature IEC-1131-3[6]. Unfortunately, there has been much controversy[7][8] and confusion generated by this effort, particularly with respect to the

inclusion of a function chart framework (Sequential Function Chart, or “SFC”) as part of the standard.

SFC is not a language, nor was it intended to be. Rather, it is a means of encapsulating Boolean or procedural code in the form of a flow chart. Yet, characterizations in the trade press have misled many into the belief that a structure and syntax exists in IEC-1131-3 for a state language.

As shown in Figure 8, taken from the draft version of the standard, SFC programs are actually composed of language elements drawn from the standard’s underlying languages. These languages, two combinatorial (Ladder Diagram and Function Block Diagram) and two procedural (Instruction List and Structured Text), are in fact the alternatives that Control Engineers have been living with for the past two decades. As such, they fail to answer the changing requirements of automation programming mentioned in the introductory paragraphs.

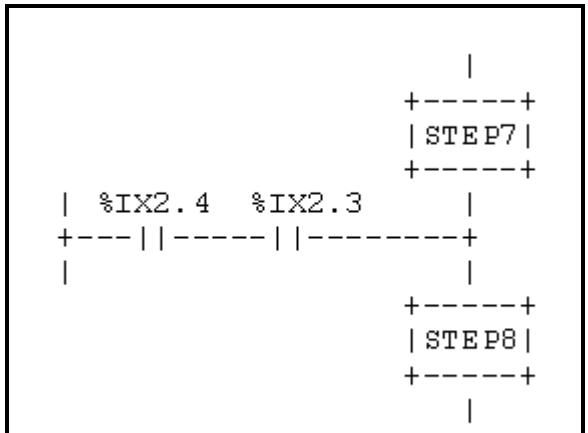

Figure 8. A sequence transition using IEC-1131-3 elements. Note that the content of blocks STEP7 and STEP8 must be expressed in one of the combinatorial or procedural languages specified in the standard.

Integrating Non-digital Functions

Many of the operations which have now become necessary additions to manufacturing operations involve such non-digital functions as servo control, analog data acquisition, and data manipulation and storage. These functions are often complex, and require embedded descriptive and parametric data. The textual representation of state language instructions allows high-level instructions to be included to accommodate these functions.

```
[125] EXTEND_ARM
    ;;; This step illustrates the use of high-level instructions
    ;;; in a state language framework.  In this case, a 'profile'
    ;;; instruction establishes a new speed for a subsequent servo
    ;;; motion.  This speed is automatically derived from a manual
    ;;; thumbwheel switch setting.
    ;;;
    ;;; The servo is then commanded to begin its motion to a new
    ;;; position.  The final instruction takes the program flow
    ;;; to the next step, but only after the servo has completed
    ;;; its motion.
    ------------------------------------------------------------
    <NO CHANGE IN DIGITAL OUTPUTS>
    ------------------------------------------------------------
    profile arm_servo maxspeed=speed_thumbwheel
    turn arm_servo to extended_position
    monitor arm_servo:stopped goto Next
```

Figure 9. Inclusion of non-digital functions in a state language “step”.

Figure 9 illustrates the use of high-level commands to accomplish servo control. In this implementation, the motion commands are in the same portion of the step as the *monitor* instruction for terminating the step. Although initially this may seem like a departure from the structure described earlier, which noted a clear separation between motion commands and step-transition instructions, there are important functional reasons for this intermingling.

```
[115] PART_CLASSIFICATION
     ;;; Here we perform a series of measurements on our finished
     ;;; workpiece using a height gauge connected to an analog
     ;;; input (named "height_gauge").  Before each measurement
     ;;; we store a new value to register "category".
     ;;;
     ;;; As soon as a test succeeds, the program instantly jumps
     ;;; to the next step, with the correct classification value
     ;;; in "category".  If all of the tests fail, the program
     ;;; jumps to the step "REJECT_PART", where the part is sent
     ;;; to the reject bin.
     ------------------------------------------------------------
     <NO CHANGE IN DIGITAL OUTPUTS>
     ------------------------------------------------------------
     store 0 to category
     if height_gauge <=bin_A goto Next
     store 1 to category
     if height_gauge <=bin_B goto Next
     store 2 to category
     if height_gauge <=bin_C goto Next
     store 3 to category
     goto REJECT_PART
```

Figure 10. Complex decision-making in a step using procedural techniques.

The most significant reason is that intermingling allows complex decision-making to be performed within a single step. For example, the series of instructions in Figure 10 call for successive values to be stored in a numeric register. After each new value is stored, a test is made of a gauging device on an analog input, testing for successively higher values. Once a test succeeds, an immediate jump is made to the next step, with the resulting classification value stored in the numeric register. Using this approach, the kind of decision-making that is typical in a procedural programming language (and often necessary in machine control) may be accomplished without the awkwardness of using large numbers of steps.

Handling Asynchronous Operations

The earliest attempts to develop a truly general automation language within a state framework found only limited application due to a single fundamental flaw. Although the vast majority of machine control applications can most suitably be cast as a state-based sequence, there are frequently multiple such sequences which must be controlled simultaneously and asynchronously on a single machine. Attempts to merge these sequences into a single linear flow of events not only result in a lack of clarity, but also may carry significant performance penalties.

Figure 11. Flow representation of multitasking.

This was resolved in the early 1980s with the implementation of multitasking in conjunction with a state language. The state language framework allows multitasking to be incorporated in an elegant and consistent manner, simply by the addition of a new instruction. In the example shown in Figure 12, the instruction *do* will cause the program flow to split into two separate paths, each of which operates independent of the other. The targets of this instruction, two steps named *load_workpiece* and *check_status*, each commence a sequence of steps which will run asynchronously to completion.

```
[1] COMMENCE_TASKS
    ;;; This step commences two independent tasks:
    ;;;   LOAD_WORKPIECE will feed a new workpiece into the machine.
    ;;;   CHECK_STATUS will examine all component feeders to insure
    ;;;     an adequate supply of materials.
    ;;;
    ;;; After these tasks are complete, the program will continue
    ;;; with the next step.
    ------------------------------------------------------------------
    <NO CHANGE IN DIGITAL OUTPUTS>
    ------------------------------------------------------------------
    do (LOAD_WORKPIECE CHECK_STATUS) goto Next
```

Figure 12. An example of multitasking implemented in a state language.

The resultant program flow, shown in Figure 11, accommodates instances where a number of simultaneous, but independent, operations must be performed. It is possible to nest such tasks further, allowing highly complex machines to be divided down and controlled in terms of their native sequences.

This mechanism for multitasking has a utility beyond the simple accommodation of parallel asynchronous tasks. It encourages the modularization of programs by creating a new organizational unit beyond the step level: the *task*.

Encapsulation: The Future of Control Language Evolution

The task construct points the way to a use of state language that makes even further progress toward the reduction of complexity. Today, state language programs may be organized such that most of a machine's functionality resides in tasks, each task relating to one mechanism or mechanical module on the machine. Using this approach, the program can become almost trivially easy to maintain.

When this programming practice is followed, the "main" program for a machine becomes a short sequence of task invocations, clearly representing the "overview" functionality of the machine. To examine the details of a given operation, you move to the appropriate task and the sequence of events for that portion of the program is stated in a clear and concise manner.

This technique, referred to as *encapsulation*, allows the complexity of the detailed control of a machine's actuators to be hidden from view when examining the "top level" program flow for a machine. This makes the program easier to understand, and acts as a navigation aid for finding the portion of a potentially lengthy and complex program you wish to examine or modify.

State languages provide a natural means to encapsulate complexity. Work is now being performed to develop higher-level, and more powerful, encapsulation tools to meet the projected demands of an increasingly complex manufacturing environment.

Summary

Changes in automation requirements and practices have resulted in increasing dissatisfaction with traditional programming methods. This has caused a reexamination of language options, and a trend toward the adoption of state language as the programming paradigm of choice. State languages can substantially remove barriers to the incorporation of new sensing and actuation technologies, better accommodate complex applications, and facilitate understanding of machine functionality among an increasingly diverse base of stakeholders.

Encapsulation techniques, used in conjunction with underlying state language programs, promise further reductions in design time and apparent complexity.

References

[1] Crater, Kenneth C. *A New Control Paradigm*. Proceedings, 1991 Industrial Computing Society Conference, pp. 545-556. http://www.control.com/

[2] Control Technology Corporation. *Quickstep Language and Programming Guide.* Control Technology Corporation, 1996.

[3] You can visit Dr. Petri's website at http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/petri_eng.html, although a more illuminating website for the investigation of Petri nets is at http://www.daimi.aau.dk/PetriNets/.

[4] Dove, Rick. *The 21st Century Manufacturing Enterprise Strategy, or What is All This Talk About Agility?* Paradigm Shift International, 1992.

[5] Martin, T. *Human Software Requirements Engineering for Computer-controlled Manufacturing Systems.* Automatica, Vol. 19, No. 6, pp. 755-758, 1983.

[6] EC1131 Part 3-93. *Programmable controllers - Part 3: Programming languages*. 1st edition. Geneva, Switzerland: International Electrotechnical Commission, 1993.

[7] Grenard, Jack, et al. *IEC-1131-3 Roundtable Discussion.* http://www.control.com/carefree/roundtables/1131.html

[8] Crater, Kenneth C. *When Technology Standards Become Counterproductive.* http://www.control.com/tutorials/language/counter.htm, July 8, 1992.

About the Author

Ken Crater is President and cofounder of Control Technology Corporation, a producer of automation controllers and software which make use of state language technology. Mr. Crater also cofounded the Industrial Computing Society, in which he served as first President. He was recently elevated to Fellow of that Society for making significant advances in the technologies of machine automation and programmable logic controllers, including his early work in the pioneering of state language for machine control.