Redesigned State Logic for an Easier to Use Control Language

Presented at the 1996 World Batch Forum (WBF) in Toronto

E. H. Bristol
The Foxboro Co., retired
Foxboro MA, 02035
Tel. (508) 543-8829
email: ebristol@mediaone.net

KEYWORDS

batch, control, language, logic, Idioms, Objects, SuperVariable

ABSTRACT

A previous World Batch Forum paper presented a language tailored to address process control problems and radically improve Ease of Use. The present paper places that paper's Ease of Use features in explicit terms, targeting a minimal 3:1 readability and top-down understanding improvement, as measured by the amount of time taken to draw simple conclusions from the visual program representation, and argues that this corresponds to **a real 3:1 reduction in total cost of the application engineering**.

To illustrate how such a significant computing practice improvement might be driven from within process control, the paper considers the systematic redesign of just one aspect of computation, control logic, based on States rather than Truth values, to improve control relevance, Ease of Use, and efficiency. While other features in the language offer far greater improvement over standard practice, logic is a generally understood area, and one of basic significance to Batch Control. Audience participation tests will demonstrate the intended improvement over conventional language logic. The paper then reviews other implications.

INTRODUCTION

Process Control frequently builds by adapting technology originating from other sources. In this way it adapted computer languages to control programming, electronic and simulation function blocks to analog PI&E control configuration, and electrical relay ladder diagrams to control logic/sequencing representation. The previous World Batch Forum paper[1] argued that the result, if generally expedient, emphasizes the incompatible **bottom-up** implementation issues over the **top-down** issues of solving the end users problem[1]; it creates a hodgepodge. That paper presented a number of representations specifically designed for the different aspects of Process Control application, top-down integration, and Ease of Use. The general focus was improved readability of the resulting control application program.

What Contributes to Ease of Use?

The interest in graphic interfaces for computers confuses the Ease of Use discussion when applied to control configuration tools. In any complex computer aided design effort there are two balanced issues in Ease of Use: the ease of creating the design, which roughly corresponds to the focus of the GUI, and the ease of readability and **top-down** understanding of the resulting design, which is the focus of good programming practice (and language design). Only after a readable language is defined are its application support tools and GUI requirements relevant.

[1] **This is new! Watch for it ( ) in the following discussions!** An ***Application-Centered Language*** is more than a higher level language. It expresses ideas from the **top-down or goal centered (detail hiding) perspective, without loss of precision**, **unlike top-down programming** which calls a named subroutine, putting off the functional definition; the function is later programmed tediously (inconsistently?) in the **old fashioned bottom-up way**.

Language design for readability is a modeling process not unlike the SP88 Batch modeling effort, but with greater emphasis on generality to serve the user's needs in detail, and on visual modeling through language structures. One of the few references[3] on language versus graphics readability[2] argues that clear application design comes from the expert's familiarity with secondary design issues, and his ability to document these effectively without confusing his readers. The language attempts to build these issues, for novice and expert alike, rigorously into the modeling process.

In general terms, improved readability comes from:

- Modeling **Application-Centered** usage and intent, de-emphasizing **bottom-up** implementation.
- Good modeling of the application hierarchy.
- Simpler reference. English allows Feed Flow, the Feed Flow Controller, the Feed Flow Proportional Band to be all derived from the Feed Flow name. There is no need for separate measurement and controller names (**F100**, **FC100**, **FC100.PB**). Later discussion will further address unnecessary name explosion.
- Keying statements by easily recognized icons, statement patterns, and formats, rather than words.
- Tabular Listing or Grouping, grouping similar elements, to support visual comparison of their related properties.
- Modeling of inherently linear relationships (sequencing, Degrees of Freedom) with ordered lines or columns of consecutive text elements.
- Automatic formatting, making clarity and predictability of the application document independent of author.

The Benefits of Ease of Use and Readability

As an overall design objective, the language targets a minimum of a 3:1 improvement in the speed with which an application program can be read. Experienced programmers generally emphasize clarity in their programs. Application engineers often fail to grasp its importance. But it benefits both reviewers and originator as he tries to understand his own work some weeks later. Basic design thinking is the smallest part of any application effort, large amounts of time being taken to debug the application and modify it to fit new requirements. This effort is dominated by attempts to understand what was done before; a 3:1 improvement in readability reflects nearly completely into a life cycle 3:1 reduction in design time/cost.

The overall impact of such an improvement is reflected in several ways:

- As the reduction in design effort itself.
- As the result of allowing designers to better understand and accommodate related designs.
- Improved access to the busy colleague, with one hour (but not three) to effectively review the effort, and participate more knowledgeably.

Review of Earlier Ease of Use Language Proposals

This section reviews the control language[1] Ease of Use characteristics. It is organized, separately, to support process and control function hierarchies. At a lower level it includes a number of specific application language modeling features:

- SuperVariable Definitions:[4]

The figure shows variables defined, as shown, with free, user-selected Attributes, grouped in tabular form. This permits Definitions which contain:

  - **Precisely those Attributes needed, independent of Vendor bias.**
  - **Duplicated versions of the same type of Attribute (e.g. multiple alarms).**
  - **Combinations of several Data Types in a single Variable Definition (e.g. a basic Real value, with support-**

STYRENE_PLANT Page: Definitions

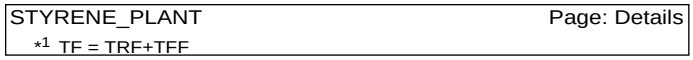
NAME	IN	VALUE	MIN	MAX	UNITS	SET	HI	LO	DEV
FST	1	_	0	800	DGC	_	780	300	20
TRF	2	_	0	100	GPM	_	_	_	20
TFF	3	_	0	100	GPM	_	_	_	20
TF	_*1	_	0	100	GPM	_	_	_	20
POL	4	_	0	300	FT	_	30	10	20
WOL	5	_	0	300	FT	_	30	10	20

[2] It summarizes experiments showing that well designed text representations can match or surpass graphics; that real application graphics are, in fact, generally hard to read.

ing operating States).

- **Clear comparative Tabular Display of similar Definitions, many to a page.**

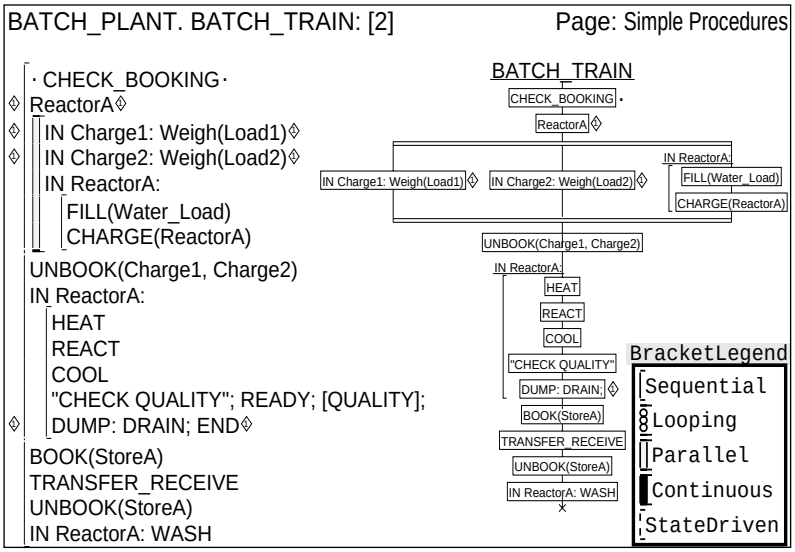
· Details Pages and Footnotes:[1]



Footnote references (the *1 in the upper figure) separate random details onto a Details Page. They simplify the listing in several ways:

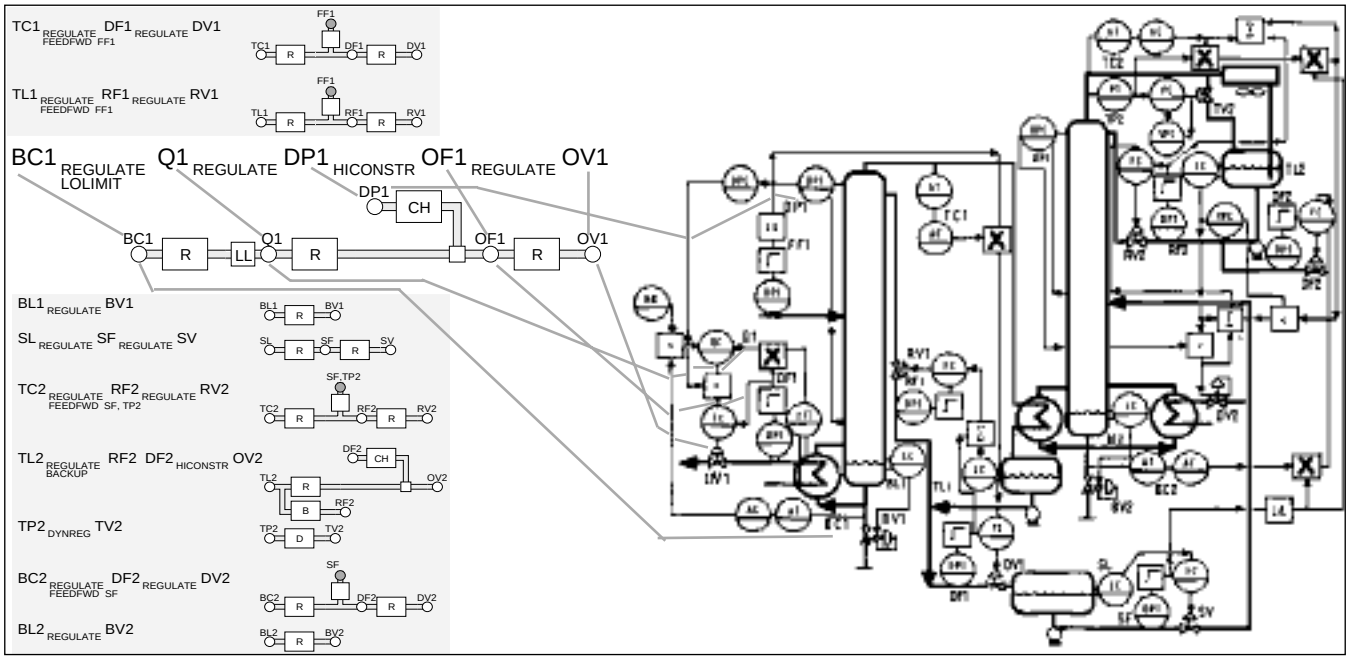
- **They allow open ended computational flexibility within the Footnote.**
- **They preserve the format of the Footnoted definition or statement.**

· Tasks, Activities, Activity Brackets:[1] Here, Bracket Icons replace conventional keywords, allowing the expression of parallel, and continuous activities within an otherwise conventional statement model. They can be illustrated by an automatically generated Sequential Function Chart. They offer the following benefits:



- **Visible structure free of hard to read keywords.**
- **The different sequenced executions, each modeled by natural sequential order, nearly as effectively as by the more graphic Sequential Function Chart.**
- **Completely general computational usage (including Continuous Controls), unlike the Sequential Function Chart.**

· Idioms and Idiom Loop Statements:[5]



An Idiom Loop Statement represents a complete, single-degree-of-freedom, cascaded loop, as a single line of text or icon elements. The statement alternates Variable names and subscripted Idiom operators, the Variables listing the variables from primary, to secondary, etc., to valve(s), including any override Variables. The Idioms[3] reflect basic or secondary control intentions (Regulation, Constraint, or Limiting) rather than their implementing computations (PID, Selector, Multiplier, etc.). The result is an expression of the basic control organization, where, as in the figure:

- **All primary controlled variables are listed, each with a single line, Degree of Freedom summary of its associated control strategy.**
- **The loops show Primary controlled variables clearly on the left of the page, and the valve(s) on the right, with the intervening goals defined from left to right. Compare the structural clarity of the Bottoms Composition to Oil Valve (BC1–OV1) statement to its PI&E expression in the figure. [4]**
-  **The statements include only Process elements, and *Application-Centered* control intentions; they imply all *bottom-up* details: parameters, setpoints, connections.**

· Theme Statements:[6]
These express specialized sequencing and coordination controls where a simple, standardized pattern applies (as in the case of cut cam followers or drum sequencers). They clarify the operation in several ways:

```
RAMP T101          START FOR VENTTIME†[1]
                   HEAT  TO COOKTEMP + BIAS, IN HEATTIME MIN
                   COOK  AT COOKTEMP, FOR COOKTIME MIN
  RAMP T102.VALUE  COOL  TO 0*[2], IN COOLTIME;
```

†[1] T102.VALUE = COOKTEMP;
*[2] T101 = COOKTEMP * (T102 / COOKTEMP)^2;

- **By modeling the usage pattern in a standard, visual format.**
-  **By emphasizing only essential elements.**
-  **By supporting general, Footnoted, computational details.**

The Role of Logic in Control

Logic entered Process Control as binary values from two sources: Boolean variables in higher level languages, and contact states used in PLC ladder logic batch applications. In contrast, Logical data is actually used to represent discrete States of the process and its equipment, or to define significant processing events by changes in the particular process States.

These States may be inherent in the equipment, such as the States of a motor (ON/OFF) or controller (AUTO/MANUAL), or derived computationally, through comparisons (such as the State of a Temperature being greater than 500°) or logical operations (such as the combined [**AND**ed] State of being under control and within constraints). Discrete States may occur in continuous calculations which control or interlock elements in the Process, or, express events, which drive conditionally executed computations. The only States of interest to a control design will thus be those which: represent process state (like motor ON/OFF States), drive continuous logical controls and interlocks, or drive conditionally executed computations.

In a sense, each **significant** State is used to cause some computed value or action. Clear designs should consider States only in this sense. The States, thus specified, will correspond to computations that are associated with different levels in the process hierarchy, thus falling into a corresponding hierarchy. The States at each level are likely to be computed from the States of lower level contained elements.

In the physical I/O hardware, Discrete state data takes the form of packed bit fields communicated between sensors or actuators and the computer. In the computing language world, these States are represented as Boolean quantities, whose binary range and 0/1, True/False values bear no relation to any natural, application State ranges or names. Enumerated typed data fits the case but lacks an appropriate computational model.[5]

A redesigned control logic ought to define:
· A computational model where meaningfully named States reflect the natural process State, and the causal role that they represent.

[3]  A Control Idiom is a basic control intent, named to reflect that intent, implemented through associated functions and a general system of application related rules.

[4] The diagram might take a couple of hours to explain, but the listing of the ten loops could certainly be outlined in five minutes. This represents a 120min.:5min. = 24:1 improvement in readability.

[5] Enumerated data computation is based on integer arithmetic, not States or logic.

- Computational mechanisms which drive actions by State and support their computation.
- Alternative, conditional actions, expressed as simply listed independent cases.

ISSUES IN THE DESIGN OF STATE BASED CONTROL LOGIC

A novel control logic, based on user named (HOT/WARM/COLD) and system (AUTO/MANUAL) States, poses equally novel challenges:

- The general invention of all aspects of a new process data type.
  - Process I/O,
  - Sequenced and Continuous State Computations,[6]
  - Real Value Comparisons, and
  - Conditional Computations.

And further complicating this:

- The State values can differ for each Discrete variable, unlike Real or Integer values which are the same for every variable.
- If the meaningful State value corresponds to its name, then field devices whose States are mapped differently need to be separately translated on line for I/O and mutual assignment.

Logic Based on States

The language represents all Process Discrete data in terms of named States. In English, the role of the State is usually taken by the adjective. If T100 were in the HOT State, we would say that T100 was HOT. Computationally, the HOT State performs a double role: it names the (HOT/WARM/COLD) data field and it names the HOT data value. The English usage shows the resulting simplification. States generally eliminate the need for named, Boolean parameters, replacing such usages as: T100.ma = 0 (Foxboro's IA) or T100.mode = 8 (Field Bus), where the ma and mode are the redundant parameter names and the 0 and 8 are the mysterious values corresponding to the AUTO State. Moreover, because the State value of an object may support many subStates and many alternative named States[7], there is no need to set aside a group of (arbitrarily named) parameters or variables to represent them.

The single process object hierarchy can include all related States, divided into broad categories (like the System and User States below).[8] The language allows the process hierarchical elements to be defined (modeled as Operations) with their associated States, as shown in the Operation Page figure.

BATCH_PLANT Page: Operation

SYSTEM STATES: CONFIGURE/SETUP/SIMULATE/OPERATE,
RUN/SUSPEND/CONTINUE/ABORT/END,
ACTIVE/ INACTIVE, BOOKED/UNBOOKED, _/INITIALIZE

USER STATES: _/RECIPE_RUN/RECIPE_HOLD/RECIPE_DONE

TASKS:

SUBOPERATIONS: Charge1, Charge2(Charge1), Reactor1, Reactor2(Reactor1),
Store1, Store2(Store1), BATCH_TRAIN:[2]

Most language entities will have built in system States inherent in the type of element; the variables defined below will have both system and user defined States. In particular, the I/O variables require the ability to define State valued data and specify its structure. Conversion between the packed physical data and the State representation is as simple as conventional floating point I/O conversions or Process I/O scaling. The table below shows simplified SuperVariable definitions of two, State valued, Process I/O variables:

NAME	DIN	STATE	*PackedValue*	STATES
C101	**1**	**OFF,LOCAL**	*0*	**OFF/ON,LOCAL/REMOTE/HOLD**
C102	**2**	**ON,HOLD**	*5*	**OFF/ON,REMOTE/LOCAL/HOLD**

The NAME column defines the name of the variables; the DIN (Discrete Input) column, the I/O card ad-

[6] Sequence and Continuous execution are distinguished by Activity Brackets, but otherwise identical.

[7] Not just the Boolean binary.

[8] The States become the point of programming emphasis, with the associated Object names and hierarchy being generally ignored, except for summary clarity.

dress; the STATE column, the current State value (containing multiple field values in this case); the STATES column, the specification for the STATE data. The extra italicized column is added to show the actual packed representation. The STATE Attribute expression lists the names of the involved sub-field/subStates separated by commas. The STATES Attribute expression lists the subState (i.e. field) specifications separated by commas, with each subState specification made up of a list of the names of the alternative State names for that subState, separated by slashes.[9]

The normal STATES specification corresponds to the natural English usage (We write: AUTO/MANUAL). The notation also allows the user to specify arbitrary bit mappings, where necessary, to accommodate peculiar hardware (or standards) conventions. For example, the example representation, 5:/LOCAL/REMOTE/HOLD,OFF/ON, allows the first subState to be packed into a sub-field defined by the mask 5 (filling in, otherwise). **Note that the specification emphasizes the application States rather than the implementation fields.** Any necessary obscurity is minimized because the specification detail relates only to low-level Process I/O conversions.

Discrete Assignments

Control language logic must support opening or closing of valves, and other simple Discrete data setting actions. State logic includes simple assignments to support this, applicable to setting the States of variables, or any other language entity having States (e.g. the BATCH_PLANT Operation). The logic assignment operator is a comma. The assignment, **C101, ON**, applied to the preceding table, causes the change (of C101) in value shown below (highlighted in reversed text). Note that the assignment only affects the relevant (OFF/ON) State field.

NAME	DIN	STATE	*PackedValue*	STATES
C101	1	ON,LOCAL	*1*	OFF/ON,LOCAL/REMOTE/HOLD
C102	2	ON,HOLD	*5*	OFF/ON,REMOTE/LOCAL/HOLD

There are several reasons for having a distinct State assignment operator in the language:

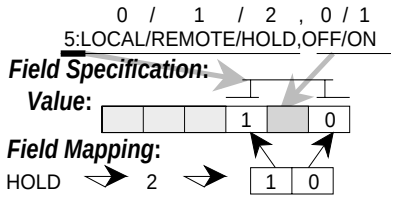
- The natural English related usage.
- State assignments in fact operate quite differently from numerical assignments.
- To operate independently on numerical data and operational States of a single variable. An assignment to a Process variable can change operational States for any associated element: Process I/O, as in **T100, ONSCAN**; or Idiom or controller, as in **T100, AUTO**.

All of this requires the described State characteristics. Multiple subStates can be set from with one assignment: **C101, (ON, REMOTE)** would cause the (highlighted) change in values shown below:

NAME	DIN	STATE	*PackedValue*	STATES
C101	1	ON,REMOTE	*3*	OFF/ON,LOCAL/REMOTE/HOLD
C102	2	ON,HOLD	*5*	OFF/ON,REMOTE/LOCAL/HOLD

Or multiple variables. **(C101, C102), (ON, LOCAL)** would cause the indicated changes in State.[10] This is use-

[9] The STATES specifications are like scaling parameters, defining associated conversions, computed by simple register shifts. The subState specifications are listed from left to right corresponding, right to left, to the sub-fields in the internal representation (making the specifications as portable as possible). Each subState representation lists the alternative State names in order of their corresponding internal translated numerical values (0/1/2/...). The figure shows how a HOLD State packed value (2) is converted using the States specification. Each field specification defines a mask (implicitly or explicitly: the 5 mask for the LOCAL/REMOTE/HOLD field), and the mapping between the (HOLD) State and its numerical equivalent. The figure shows how that value is shifted (either collectively, or, as in the figure, in terms of the individual bits) to fit into the packed format. The reverse mapping would convert the packed form to the State name.



ful to concisely set groups of similar States at the start of a new operational phase.

NAME	DIN	STATE	*PackedValue*	STATES
C101	1	ON,LOCAL	*1*	OFF/ON,LOCAL/REMOTE/HOLD
C102	2	ON,LOCAL	*3*	OFF/ON,REMOTE/LOCAL/HOLD

State based assignments can assign values between State valued variables. The source (second) variable is parenthesized:[11]

C101, (C102) .

This affects all associated States.[12] A more restrictive form explicitly states the permitted affected States:

C101, (C102(OFF/ON)) .

Simple Conditional Computations; State Prefixes

Similarly, we require a simple mechanism for conditional execution of statements, the State Prefixed statement:[13]

AUTO: C103, OPEN

It expresses the setting of **C103** to the **OPEN** State if the associated Operation is in the AUTO State. Appropriate Process hierarchical State definitions will minimize the need to include logical State combinations, and the State Prefixed statement is designed for the simplest case. However it does include the ability to test for several subStates applying simultaneously (the comma operator, analogous to an AND) or alternatively (the slash operator, analogous to an OR).

AUTO,REMOTE: C103, OPEN

AUTO/REMOTE: C103, OPEN

The comma and slash operator usages reflect the same English based conventions used earlier to express compound STATE values and STATES specifications. They are **not Boolean operators**, expressing the States which will permit the conditional, statement-continuation execution. The State usage can minimize complicated and confusing logical computations: The NOT operator will normally be unnecessary. It is clearer to say something is COLD, than to say that it is NOT HOT.[14]

The State Prefix examples have been based on States of the associated Operation. The use of more complicated State computations with other language entities requires the Environment Declaration described next. The two shorthand expressions included here will later be seen as simplifications of that notation:

[T100] AUTO: C103, OPEN

[T100>500]: C103, OPEN

The first line allows testing of States drawn from the square bracketed variables. The second allows the

---

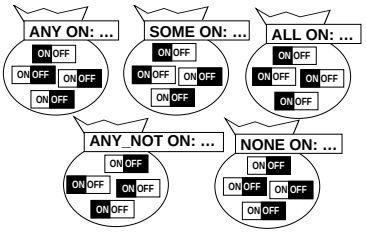
[10] Note that the properly implemented assignment applies different translations to differently specified variables.

[11] Making the sink variable the main focus of the statement.

[12] Including not only user defined States, but I/O States, controller States, etc.

[13] The counterpart of the normal IF statement in State based logic, without the IF noise word, making the application more readable (particularly when automatically formatted).

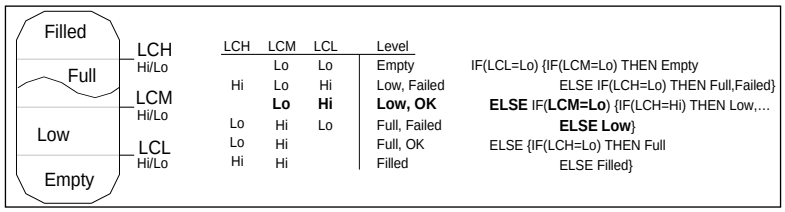
[14] The State Prefix can include a NOT operator when this is made necessary by a subState having more than two alternative values: as in: **NOT HOLD: C103: OPEN**. "**NOT HOLD:...**" will normally be clearer than "**LOCAL/REMOTE:...**". The language also permits several other, similarly used, keyword operators (illustrated on the right): ANY, SOME, ANY_NOT, ALL, NONE, and ELSE. Except for the ELSE operator (which is used in its normal meaning), these are meaningful only if several variables, with their own independent States, are being compared, under the Environment Declaration described later.



normal numerical comparisons.

Problems with More Complicated Logical Computations

Simple assignments and IF statements will generally be clear and readable. But complicated combinations of conditional statements and computations present special problems. As shown later, the language uses the State representation to convert all complex conditional statement combinations to a Case statement based form, and all Discrete computations to Truth Tables. The figure below shows the computations for converting low level (level contact) states into higher level (Tank Level) States. The associated computation is expressed as a modified Truth Table and as a complex IF statement. The figure makes the difficulties of the latter computational form apparent.



The Truth Table is modified to allow lines to be entered in any order and to allow blank "Don't Care" entries. Individual column entries can include composite States (HI,ONSCAN) or alternative States (LO/OFFSCAN).[15] The Truth Table is generalized to return that result corresponding to the top-most line with consistent input variables.[16] This allows the Don't Cares to be used to rule out successive cases, top to bottom.[17] The later combined Truth Table/Case statement form shows how the inherent symmetry of Truth Tables and Case statements, and the distinct separation of variable selection, Discrete computation, and choice of statement cases clarifies this kind of computation.

Intermixed sets of IF-THEN-ELSEs raise one other issue: The States being set and the States being tested can easily get tangled up so that the test is affected by conditionally executed statements within the tangle, whether or not intended. In the language, the tested States represent snapshot values taken before any conditional action takes place.

Case Environments and Environment Declarations

Conditional computation combines any selection of variables, logical computations and conditional response actions together into the single (IF) statement. The language clarifies conditional computations, separating elements so that the essential action is encapsulated in the State Prefixed statement cases. The Case Environment mechanism controls the variables (whose States are to be tested), with its Environment Declaration. The figure below shows a Case Environment based on the above Tank example. The overall effect is to set up a test that fills the tank and terminates when it is properly full, but that closes the valve if the tank is overfilled:

```
PROCESS. TANK                                    Page: Pr

TEST_TANK
   [LCH; LCM; LCL];
     LOW/EMPTY:  TKV, OPEN;
◇    FULL:       END;◇
     FILLED:     TKV, CLOSE
```

The Environment consists of all statements separated by terminating semicolons (in this case, all statements bounded in the Activity Bracket). The topmost statement is the Environment Declaration. It consists of the square bracketed list of contact variables separated by semicolons. All State Prefixes in the Case Environment will be applied to the collection of all States derived from the variables included in the Declaration.

The result is a Case statement structure, framed by the Declaration and the separating semicolons, and

[15] Or as in the figure, composite output States: **LOW,OK**, etc.

[16] State combinations may arise, not matching any table entry, computing no result.

[17] The example uses clear inconsistency to diagnose contact failures and guess results.

based on the collective States of the declared variables. The Prefixed statements define separately what would happen if the States LOW/EMPTY, FULL, of FILLED applied. There is a catch in the example. If the States in the State Prefixes were in fact the States of the declared variables, the combined expression would represent the complete computation. However, those States are all drawn from the set HI/LO; the States in the case State Prefixes are the States of the Tank (EMPTY/LOW/FULL/FILLED), which need to be computed. Instead, an associated Details Page Truth Table must be set up to compute the needed States:

PROCESS. TANK

LCH	LCM	LCL	Result
	LO	LO	EMPTY
HI	LO	HI	LOW, Failed
	LO	HI	LOW, OK
LO	HI	LO	FULL, Failed
LO	HI		FULL
HI	HI		FILLED

This is the previous Truth Table, but with the system variable, Result, set up to pass the State data. If a programming error is made it will be isolated as an error in computation of the States, or, less likely, an error in the choice of conditioning States. It will not, in any case, be confused, as would be the equivalent intertwingled IF-THEN-ELSE. In any Case Environment, its State Prefix conditions are always based on the State values which applied at the time that the Environment Declaration is first encountered (before any case is executed). This prevents any possible confusion by States reset in the computation.

The Environment Declaration mechanism allows comparison of Real values. Within the normal Declaration semicolon separator, a Real variable, followed by a comma and value indicates a comparison to that value: **[F100, 500; P200, (P300)]**. The result of the comparison is expressed as a HIGH/LOW/EQUAL[18] State value, used like any of the other States.

Truth Tables (Three Forms)

The Truth Table can also be used for independent logic computations. The language supports three forms, the first acting as above. These can be entered as statements anywhere in a program, either to be run once in sequence, or to be run as continuously executing logic. The first form runs as described above.

The second form, a ladder variant shown below, combines sections of Truth Table in parallel. The Valve

FUELTEMP	TANKTEMP	VALVE
HOTTER	HOT	OPEN
	PRESSURE	
	NORMAL	

FUELTEMP	TANKTEMP	VALVE
HOTTER	HOT	OPEN
	PRESSURE	CLOSE
	NORMAL	

will be placed in the OPEN State if either of the two parallel paths (TANKTEMP HOT or PRESSURE NORMAL) and the FUELTEMP HOTTER State applies. The left hand version only opens the valve (because there is only the one line in each of the columns). But the right hand version has a second entry in the Valve column, with assumed Don't Cares in the remaining columns. If the opening condition fails, this second line applies and the valve is closed. In certain cases, the ladder logic variant is more easily read.

The final variant, used in special cases, is shown below: the output variable is replaced by an action statement.

C101	C102	C103	ACTION
OFF	OFF	OFF	L100, OFF
OFF	ON	ON	L100, ON; L101, ON
ON		ON	F103 = 50

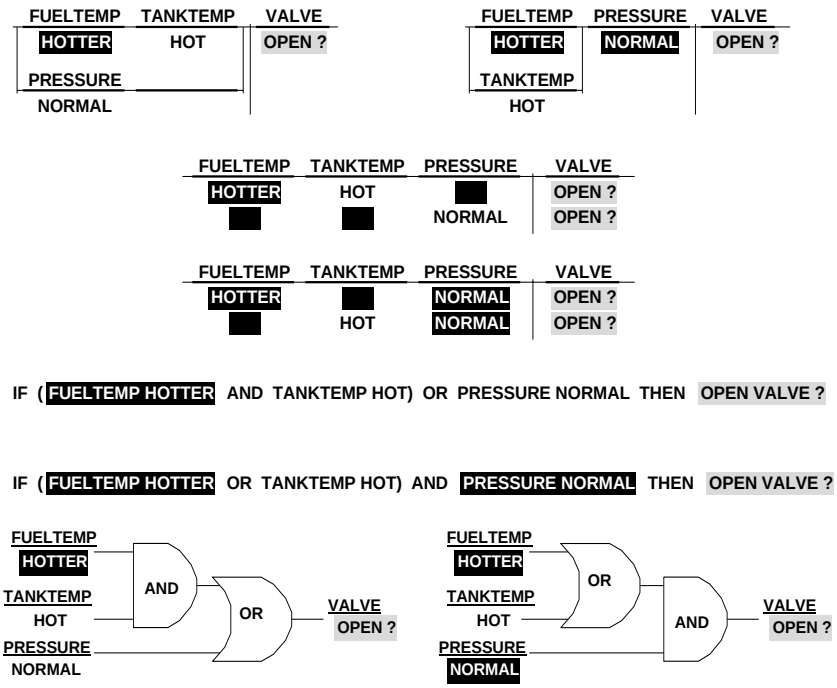
Ease of Use Proved

The discussion relies on judgment and experience to demonstrate Ease of Use. But human factors issues can only be validated experimentally. There have been Process Control Ease of Use tests based on benchmark applications.The trouble with these tests is that their results are ambiguous: Does each of the benchmarks fairly apply to the purpose of each tested system? And are they simple enough to test the system characteristics alone without testing the IQ or the manual dexterity of the user?

[18] The HIGH/LOW/EQUAL spellings disambiguate them from the alarm States HI/LO.

Tests about simple language tasks and yes/no questions based on simple reading of the syntactic form eliminate this ambiguity. Only the reading response is tested, not IQ. The figure below shows two logical computations represented in the ladder and basic Truth Table forms, and in IF-THEN-ELSE and Logic Diagram forms.

In each case, the reverse lettered States are assumed to apply (including the everywhere assumed Open Valve output State). In each case, the question is: Is the statement, with the States so assumed, consistent with the Valve Open action? When these kinds of tests are given, one statement at a time, to individuals or groups, and the individuals are asked to indicate (Yes or No) whether each statement is consistent, the time taken represents a measure of the ease of reading of that kind of statement form. In general, in such a test, the Truth Table forms are most easily read, with the IF statements taking up to three times longer, and the logic diagrams falling somewhere in between.[19]



The Logic Diagram results are interesting because of the strong European preference (over ladder logic). Logic Diagrams display signal paths. They are essential to the implementation and debugging of large collections of traditional continuous logic circuitry, but do not clarify the State based cause and effect called for by the application and language.

DETAILS; ISSUES OF AN EFFICIENT IMPLEMENTATION

Note: the discussion below describes material programmed by a system programmer in the efficient implementation of a language system; not anything ever seen by any user!

The language targets a 3:1 improvement in readability, with efficiency comparable to conventional technology. The language design envisions a compilation into a low level, back-compilable, portable code. The discussion addresses assignments and Case Environments. The compilation creates a two level code, in which the computation is broken further into compiled pieces which carry out the needed action.

Discrete Assignments

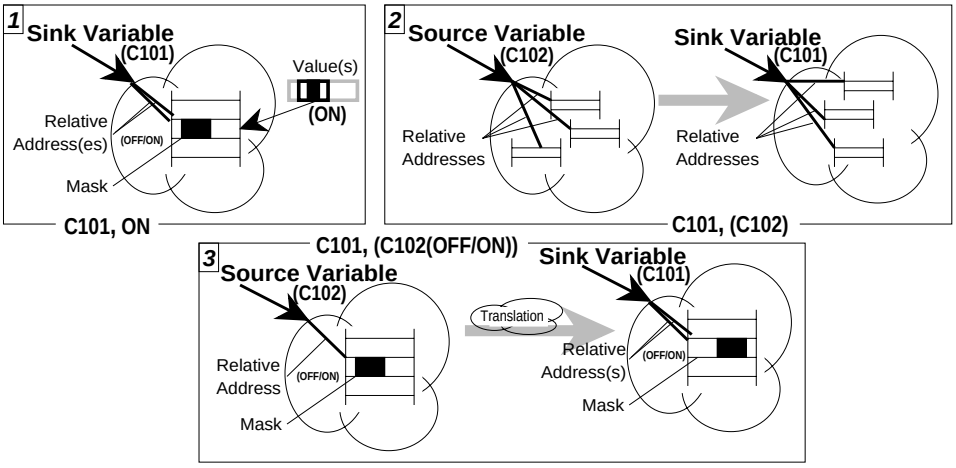
As indicated earlier, States are internally represented as packed integer data fields. A State is represented by its location, value, and mask. The States of a variable may be drawn from several such packed integers and so the complete value may require a list of such representations.

Three distinct assignment cases are illustrated:

1. The simple assignment of one (**C101, ON**) or more (**(C101, C102), ON**) variables to a simple or composite State.

[19] This makes sense because the Truth Tables reduce the cause and effect analysis to tracing of the associated lines, the IF statement must be parsed and then analyzed, and the logic diagram expresses the parsed result but it must still be analyzed.

This assignment will be compiled into a address-mask-value (or list) representation of the State (**ON**) followed by a reference to the sink variable (**C101**), or a list (**(C101, C102)**). Differently mapped variables will be treated separately.



2. Assignment of all States between identically State-structured variables (**C101, (C102)**).
   This assignment will be compiled into a reference to the source variable (**(C102)**) and list of the relative addresses of all its State containing integers, and a reference to the sink variable (**C101**) (and its corresponding list).
3. Assignment of a selected field between one variable and one or more differently structured variable(s) (**(C101, (C102(OFF/ON))**).
   This assignment will be compiled into a reference to the source variable (**C102**), and the relative address and mask for the intended State (**(OFF/ON)**); and a reference to the sink variable (**(C101**), a translation table defining the translated integer value for the sink variable State corresponding to the source State value, and the relative address and mask for the sink State.

In each case the resulting compiled data is sufficient to compute or back compile to the original statement form.

Case Environments, Case Statement Forms, and Truth Tables

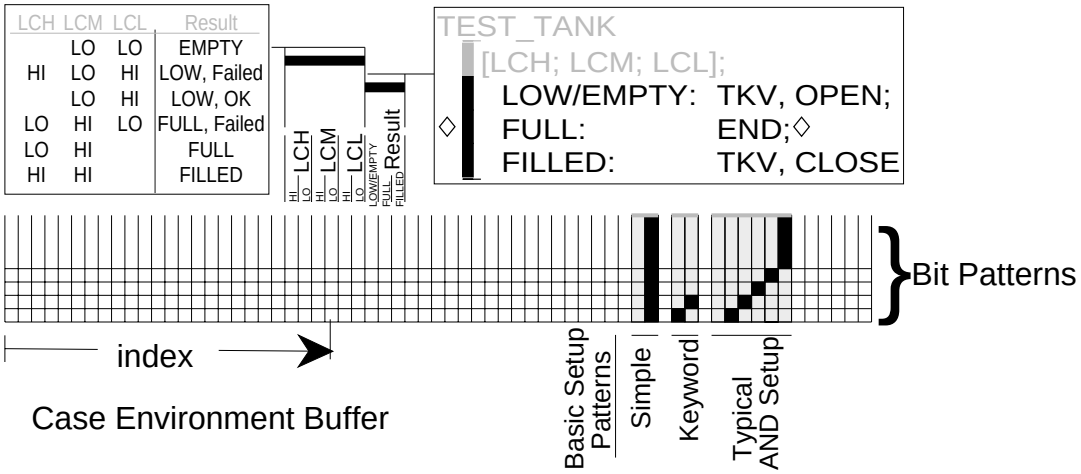
The various Truth Table and Case Statement implementations are derived from the same basic execution pattern: Any Environment Declaration (or Truth Table Heading) corresponds to the next available group of consecutive bit pattern elements in an Environment Buffer array (figure below), and each distinct State Prefix (or Truth Table row column entry) expression corresponds to a particular element in the array. The Declaration (or Heading) is compiled into codes which compute, for each of its variables, the successful or failed match (of each expression State) for each relevant pattern. Each State Prefix (or entry) will be compiled to include the index of its bit pattern, permitting easy testing of the computed bits. Pattern elements are allocated from the buffer as needed.[20]

The arrangement of patterns is illustrated in the figure below for the earlier Case Environment. In this case, the corresponding Truth Table must be executed first to compute the system variable Result. Since the Truth Table uses the HI and LO States of each Truth Table variable in its computations, there is a bit pattern for each. From the Result values, the three Case Environment State Prefix expressions can then be computed; there is also a bit pattern for each.

The figure shows the patterns as they are implemented, with separate Truth Table and Environment patterns.[21] This structure supports the interpretation of Tables and Environments, allowing each to include nested definitions of other Tables or Environments. The structure is sufficient for efficient execution and back compilation.

[20] They will be released, stack-like, for reuse when their Environment has been completely computed.

[21] In actual practice, the Truth Table States would be completely computed before the Environment patterns were needed; thus the Environment patterns could overwrite the Truth Table patterns, stack-like on the Buffer. The figure shows three basic pattern types, each arranged to efficiently support a particular kind of Prefix or entry.



CONCLUSIONS

The earlier paper argued for a single general control language, capable of representing Batch Control in the sense of the SP88 model, but also capable of supporting the other forms of Process Control. Such a language would reduce the artificial boundaries that have been imposed by the different sub-disciplines of Process Control, and eliminate the hodgepodge of control data bases and representations that now must be reconciled. Moreover it would present the technology of the different sub-disciplines in ways that were helpful to each other.

The resulting language calls for innovation in each of the representations, to make the corresponding sub-discipline easier for the expert, and accessible to the novice. This represents an advance for the expert even though he has to relearn his strategies because he is now better able to communicate his designs and leverage the personnel around him. At the same time, such an improvement in readability and Ease of Use should have a comparable and measurable effect on life cycle application design cost.

Within the Process Control environment, where even FORTRAN is considered hard to use, such a proposal is highly suspect. Accordingly, this paper has taken one Batch relevant sub-discipline, logical control, and opened it to detailed discussion, showing how Ease of Use in a language can be analyzed, designed, and tested, and how such radical proposals need not be incompatible with efficient Process Control.

Bibliography

[1] E.H. Bristol, “Not a Batch Language; A Control Language”, World Batch Forum, Newtown Square, PA, May ‘95; also ISA Transactions, Dec. ‘95.

[2] E.H. Bristol, “Computer Language Structure for Process Control Applications, and Translator Therefore”, U.S. Patent No. 4,736,320, Apr. 5, ‘88.

[3] M. Petre, “Why Looking Isn’t Always Seeing: Readership Skills and Graphical Programming”, Communications of the ACM, June 1995, pp. 33–44.

[4] E.H. Bristol, “SuperVariable Process Data Definition”, ISA SP50.4 Working Paper, Oct. 24, ‘90.

[5] E.H.Bristol, “Rules, Statements, and Idioms”, 17th Annual Advanced Control Conference, Purdue University, West Lafayette, IN, Sept. 30 – Oct. 2, ‘91.

[6] E.H. Bristol, “Sequencing, Logic, and Theme Statements”, Annual AIChE Meeting, Chicago, Nov, ‘91.

[7] E.H. Bristol, “Standardizing Application Language Systems”, Chemical Engineering Progress, Aug. ‘84, 65–70.

