Intent-Based Process Control Configuration Models[1]

E. H. Bristol 12/16/99
The Foxboro Co.
Dept. 0318, Bldg. C41-1H
Foxboro MA, 02035
Tel. (508) 549-2019
Fax: (508) 549-4379
email: ebristol@foxboro.com

KEYWORDS

Future Technology, Information Models, Idioms

ABSTRACT

Process Control application engineering would be substantially less expensive if supported with readable self-documentation. An important advance would be models and languages designed to more clearly represent application Intent (as formalized herein), without the usual implementation obscurity. This paper analyzes how Intent, in the sense later defined, can be usefully defined and expressed. Process Control has traditionally been defined in terms of a number of automation levels. This supports the intended Intent concept in one way, defining the implementation of higher application goals in terms of lower level ones. But a different, even more useful model is the leveling of physical sciences where each level of problem is best addressed by an appropriate kind of theory, itself dependent on lower level theories. Each theory must be proved by more fundamental theories even though those theories are much too complex to address the higher level problem directly. But more than leveling is needed. At each application level, the associated language should support concepts that make its normal usages clear. It should ensure that appropriate application practices can be expressed transparently in terms of their Intent. It should allow the engineer to clearly relate the result to the expected implementation, allowing him full control over the application. The paper will expand on prior papers to show more generally how these concepts can be developed.

INTRODUCTION

This paper discusses the problem of programming and documenting complex control systems so that any reader, normally familiar with the process, can pick up the design and understand it directly. This is a general issue in software systems, which has frustrated much process control application. Even our best application models don't always help as much as they should. The problem expresses itself in many symptoms:

- While a page of English is easy to read for its literate users, a page of C language source code is much harder to read for its experienced users.
- Despite our best intentions, large systems projects often create results completely unrelated to their original intended goals.

[1] Based on "Intent-Based Process Control Configuration Models" by E.H. Bristol, published in the proceedings of the ISA TECH/1999, October 5-7, Philadelphia, PA".

- We try to make things simple and make them inflexible instead; we try to make things flexible and we make them impossible to use.
- We try to make a system well structured and we make it cumbersome.

A key modern idea within modern computing technology is Information Modeling.[1][2] It fits with the new ideas of Objects.[3] It should be the basis of most of our broad systems and standards work, and yet we still seem to be missing some dimension. On the face of it, Information Modeling is about modeling particular systems. However, the Process Control problem is not just a single application which can be modeled and explained once and for all but a broad family of applications, requiring a general model of the shared aspects of the members of the family and the strategies for developing them.

The paper develops a concept that we will call Intent-Based Modeling. The notion is that a Process Control system is made up of elements that are best described by their Intent (as formalized in the following section), as seen at the application level and the domain in which they are defined. Each domain exists in relation to other domains. As illustrated in Figure 1, these domains may exist in a hierarchy (shown on the left), each depending on lower level domains to implement its Intents in terms of the lower level concepts. Or they may include mutually supporting elements in parallel (shown on the right).

Process control has often been discussed from a leveled point of view. But the physical sciences, particularly physics, as in Figure 2, lend themselves to an organization, more like the intended Intent structure. Typically each layer in the physical sciences is defined in an entirely independent, self-consistent way. But the lower level sciences are used as the foundation to the higher level one, in such a way that each level is formally proved within the next lower level. On the face of it, only the lowest level is then necessary, but the higher level discourse would be completely incomprehensible if discussed directly in terms of the lower level concepts. Issues or analyses, presented at the right level, become clear, or at least manageable.

The subroutine hierarchy, of Figure 3, is similarly built by using subroutines in a lower level to implement the higher level subroutines. The subroutine permits the reuse of lower level defined functions. It also imposes a type discipline on the choice of data connections made to subroutine calls. Objects extend this discipline and the earlier function block language capability to a data structure environment with richer Object messages or connection type discipline. But none of these support the specialized physical-science-like domains of internally consistent relationships of the kind that we need.

Figure 4 illustrates the way in which Objects can model a cascaded structure of process Controllers. Objects can be designed to discipline the connections between controllers so that only controller inputs could be cascaded from controller outputs. But more is needed. This discipline, like other typical computer language disciplines, is only local in its impact. For example, it provides nothing to prevent a cascade that loops back on itself, as illustrated, however absurd this is in practice. What is needed is a more global kind of discipline, coupled to a statement of the structure or user intentions to be supported at each level. Such a structure has many benefits:

- It supports easily understood application documentation.

[2] Footnotes will be numbered, simply super-scripted, while references will be included super-scripted in brackets.

[3] In the sense of Object Oriented Programming. The paper will capitalize terms, drawn from Standard English when these terms, however suggestive, are taken in a specialized technical sense.

- It unifies practice, encouraging the use of single, integrated, best-choice strategies for implementing related Intents.
- It supports the coordination of related functions. The later discussion[2] shows how the same Intent formalism could be used to generate both the controls and the related human interfacing GUI.
- Eventually, it supports true “artificially intelligent” control configuration, based not on magic, but on the rational development of standard Intents with standard practices for implementing them.

This kind of formalism is easiest to see in the software domain. But because the different software formalisms have no clear connection to what we would recognize as application issues, they cannot reflect this kind of Intent. The author has proposed a process control language[3-5] that illustrates some formalisms at the process control level, particularly for feedback control, from which the above cascading example is drawn. Progress in development of higher level thinking about process control will be measured by our ability to create such formalisms.

WHAT IS INTENT?

Intents of a control system are inherently developed in distinct application domains or layers, where a set of goals in one domain is translated into goals in other more technical domains. Such a structure is common place to a design planning discussion. But in order to be useful as a means of clarifying standard discussions or application designs, it is necessary that each domain have an associated formal practice of goals and constraints on the implementation of those goals. Further, the formalism at each level requires at least an informal statement of the expected translation between its goals and the goals in related domains.

What is not Intent?

The proposed concept of Intent is based on realistic engineering. In a control design, the Intent to “Make Money at the Push of a Button” is either at too high a level to support a meaningful engineering practice, or an appeal to Aladdin’s Lamp. The Intent to achieve 200% efficiency is ridiculous physics, also not meaningful.

What is Intent?

Each relevant application domain should support a set of well-defined Intents having the following characteristics:

- Each Intent has a name that can be directly understood by any normal reader of the application documentation without any need to puzzle over what is meant when that Intent is invoked on processing elements. In this respect the name reflects application Intent not computational form. It may be necessary to have distinct Intent names for functions, whose implementations are similar, because without them the Intent would be ambiguous. **A well-selected name is crucial to clear control documentation.**
- Intents fit within a specialized notation or graphics, which expresses a generalized and internally consistent set of relations between them. This formalization should enhance Intent visualization, eliminating ambiguity. Ideally any notations should support tabular listing or similar naturally readable forms without loss of flexibility.
- They reflect well-understood common practices.
- There are well-recognized strategies for implementing each Intent and relating it to the other application domains when applied to the process elements.

- The notation relationships reflect significant structural implications in the application domain well enough to either prevent the expression of inappropriate application designs or make them obvious (e.g. eliminating the Degree of Freedom loop in the earlier control diagram). These relationships and strategies distinguish the implementation of an Intent from the conventional subroutine or Object.

  Note: that the reflected relationships may be quite simple in themselves (like the rules in the mathematical notation below). Their value is in clarifying and enforcing the proper behavior in the large number of cases occurring in the real application.

- The notation reflects the underlying implementation considerations well enough to support automatic compiled implementation in which requirements derived from the Intent statement and obvious to the engineer, need not be redundantly expressed in different system elements (e.g. in configuring both control structure and operator interface).

INVENTING A NOTATION; ALGEBRAIC SYNTAX

The simplest familiar example of global structuring and syntax is the traditional algebraic equation. If we did not already know of such a notation we would recognize that it would somehow be about combining two algebraic quantities computationally. In this situation both the nature of the computation and some separation of the arguments are needed. Making a symbol for the computation do double duty makes sense: **a+b**. Of course, if the argument symbols are restricted to a single character, at least one of the computations can be represented by an empty character: **ax**. Although there are a number of equivalent standard notations, this usual one has a further advantage: It keeps the computational symbol in the middle of the expression where it is most easily related to its arguments and remembered.

Figure 5 shows standard expressions. In its traditional presentation, algebra is developed progressively to represent the following mathematical concepts (Intents):

- Linear expressions adding variables with weighting coefficients.
- Polynomials, also based on terms with coefficients.
- General expressions.

In this usage, the addition and subtraction operators are most frequently required (as lowest precedence operators) combining other more complex terms. Parentheses can then be used to resolve more unusual cases. This gives rise to the traditional operator precedence and form. The result is familiar to all as a way, impossible with natural language, of easily capturing the underlying mathematics.

Figure 6 shows how the basic algebra has been extended in programming and programming assignments. We can compare the readability of the algebra to its programming counterpart. (The short C program converts a string of hexadecimal characters to their numeric equivalent.) The program adds novel kinds of operator as well as the familiar control structures, and combines these in multiple lines.

But there are deeper reasons why the algebra is simpler, based on the kind of Intent based representation developed in the paper:

- The simple alternation of the operands and of actual or implied operators.
- The equations each represent single independent ideas.
- Each deals with a single kind of real valued data.
- Each describes a simple computation in terms of its intended intermediate results, and the data from which they are to be computed.

An experienced reader can recognize quickly whether a line of algebra is correct or meaningful, if he is aware of the subject matter, as contrasted with the C program. This is not a reflection on C as a language. C is a general-purpose language intended to represent complex calculations about things whose Intent is unrelated to the programming form. The program does not describe Intent but lower level computation whose Intent is only remotely related to its use of algebra:

- The bottom line program assignment is like the mathematical equations, but it discusses numbers being used to implement an internal form rather than some intentional view of the user's (hexadecimal) data.
- The same is true, in a less direct way, for all of the other statements. They all represent an implementation of lower level concepts that are intended to mix in ways determined by the user but not tied to any predictable Intent that could be instantly recognized by an outside reader.[4]
- The program mixes a number of different representations for the data: A letter can stand for a variable (c), or in single quotes ('A') for its numerical equivalent (which itself can be represented numerically as well).

INTENTION MODELS OF CONTINUOUS CONTROL; IDIOMS

The best example of an Intent notation is based on the control Idiom.[3-5] As illustrated later, this is a generalized control operator which:

- Is based on a standard continuous control role or Intent.[5]
- Computes control actions applied to output Variable targets from input and output Variable data.
- Derives target and state data attributes from the Variables implicitly, without formal attribute connections.[6]
- Recognizes control Degrees of Freedom paths, and path changes in the face of external failures or imposed limitations and constraints.[7]

When automated, this structure would allow individual Idioms to function autonomously:

- Supporting the commands or target values that they received by deriving commands or target values to be passed to their downstream Idioms.
- Recognizing any failures of the downstream Idioms to meet targets, adjusting its own operation to accommodate in the best way possible.
- Allowing controllers to continue to operate properly without regard to the action of the other controllers' occasionally conflicting objectives. Excepting the niceties of controller tuning, each controller can thus be positioned and commissioned independent of interactions with its neighbors.

This concept automation generalizes many standard practices, such as anti-windup and blend pacing, in a single Intent model.

---

[4] This is not a limitation of the C language, but a necessary characteristic of any general purpose language, designed to allow its writers to express arbitrary programs, dealing with arbitrary data forms in a predefined notation.

[5] A Primary Regulation or Constraint control role, a more complex Multivariable Blending or Decoupling role, or a Secondary Feedforward or Compensation role.

[6] For example, the notation refers to the control Variables alone, relying on the underlying relationships to distinguish the roles of measurements and setpoints, implicitly.

[7] The Degree of Freedom path addressed here represents the progression of controlled variables taken in a typical cascaded control structure, from a Primary variable, through successive Secondary variables, to the final manipulated actuator or valve. It represents a progression of allocations of measured variables leading to a final manipulated variable, which imposes the single Degree of Freedom utilized by the path. Any constraint controls introduced in that path change the path when they become active, replacing the more Primary variables by the Constraint variables.

In the example loop expression, $\mathbf{T100}_{\mathbf{REGULATE}}$ $\mathbf{P100}_{\mathbf{HICONSTRAIN}}$ $\mathbf{F100}_{\mathbf{REGULATE}}$ **V100** (which is conventionally expressed in the Figure 7 cascaded loop diagram):

- $\mathbf{T100}_{\mathbf{REGULATE}}$ represents the regulation of **T100** by manipulating the downstream variable (the setpoint of **F100**).
- $\mathbf{P100}_{\mathbf{HICONSTRAIN}}$ represents the high constraint control of **P100** by overriding (when necessary) the manipulation (of **F100**).
- $\mathbf{F100}_{\mathbf{REGULATE}}$ represents the regulation of **F100** by manipulating the valve **V100**.

The notation abstracts the representation of well-understood standard practices for implementing cascaded control loop structures. The **REGULATE** and **HICONSTRAIN** Idioms are each implemented by PID controllers; the distinct names disambiguate the distinct Intents. The interpretation of the statement distinguishes the underlying roles of measurements, actuators, and setpoints, without requiring their explicit reference. Its linear form not only permits easy distinction of appropriate from inappropriate usage, but also actually prevents the kind of looped back cascaded structure shown in the Introduction. It easily supports the distinguished use of measurement and actuator variables.

Not only is the control structure easily compiled from such a form, but appropriate operator interface displays could also be compiled, as illustrated in Figure 8.[2] Such an interface would be more than a compilation of the individual controller faceplates; it would capture the current operating modes and override conditions as indicated by shading the inactive Constraint controller faceplate and drawing though the connection between Primary and Secondary controllers. The compact notation has other advantages, such as allowing the expression of continuous controls to be merged with all of the traditional computer language practices.

ON THE DISTINCT NAMING OF INTENTS

Effective Intent notation critically depends on the use of Intent names that are both meaningful and appropriately general. This can be illustrated with the Idioms. The Idiom name is a general operator symbol that must lend itself to general use. This permits a system to solve many applications with the fewest possible distinct concepts. At the same time the Idiom name must reflect an Intent that the user can immediately distinguish from other distinct Intents.

Furnace combustion fuel/air mixing control illustrates the issues nicely. Figure 9 shows the typical cross-constrained fuel and air control loops. But this decomposition does not provide a clear statement of the Intent, which is: to ensure that the fuel is never allowed to accumulate, as a safety hazard, in excess of the air. The obscurity is worsened when the elements of the technique are mixed with other constraints and controls. Thus a distinct Intent module, named like Combustion Control, is needed to draw the reader’s attention to the special flavor (and hide the implementation detail). Of course this doesn’t adequately name the generalization of this technique, which is used in other contexts where it is unsafe for one component in a blend to accumulate un-reacted in excess. While process control people at least recognize the technique and generalize its use appropriately, a generalized name like Safe-Blend would perhaps be clearer, at the same time matching hokey industry-naming traditions.

A similar issue arises with the advanced multivariable controls, currently going under some variation of the name Predictive Control. On the face of it, these techniques would clearly embody the Intent structure in their objective and constraint functions. But the dependence on a colorless generalized mathematical constraint modeling structure tempts the user to use tricks to accomplish constraint conditions (like the above Combustion Control) which would otherwise be expressed more clearly by specialized Intent specific techniques. In this case, the monolithic combination of multiple, independent, user-constraints also confuses any reader of the application documentation (unless separate documentation spells out the real goals).

The problem is readily apparent in the related difficulties seen in designing meaningful operator interfaces for such controls. Let it be said: this has nothing to do with the operator's lack of sophistication; the operator's requirement and perspective are much more fundamental to the application than the mathematics. These designs would be clarified by classification of Predictive Control usages into more application meaningful groups and decompositions, in terms of their distinct constraint usage.[8] They could then be taught as distinct applications to their design engineers, and tailored for clearer operation. As above, this classification could lead to higher levels of "artificially intelligent" configuration, operation, and display.

LOGICAL CONTROLS

Logical controls also operate with continuous time data. The major difference between normal continuous control and logical control is the different data type and the absence of traditional feedback control. Logical controls still follow a Degree of Freedom path from some Primary command switch state, through a progression of override constraint checks and fanouts, to the final set of contacts which may power a motor or other process control support machine. They could be the basis of a set of Logical Idioms. However, the field has not developed that kind of practice. In this case, the language[5] uses simpler mechanisms to support Intent at a different level.

Traditional computer languages use Boolean True/False or 0/1 values for representing discrete data. Nothing could be more destructive of the readability of application programs. The proposed language replaces these with multi-valued States: **START/STOP**, **OPEN/CLOSED**, **ON/OFF/HOLD**. The Intent and meaning of the different States is thus spelled out in a manner consistent with the current level of practice. Computer languages have taken on the enumerated data type, which is similar except for being computationally tied to integers.

The proposed language uses truth tables and variant ladder diagram forms to represent the computation without compromising the non-numeric nature of the computation. These forms do not add to the expression of Intent, but they allow easier following of cause and effect between the different States and Events in its implementation.

SEQUENCING

A still more basic form of computation is Sequencing. Unlike continuous control, the implementation of sequencing and its underlying Intent are usually clearly related to each other. For this kind of application, a subroutine-like capability,[9] defining higher level tasks in terms of sequentially executed lower level tasks, provides a good basic Intent notation. There are variations of sequencing which lend themselves to improved notation, distinguishing basic sequencing from:

- The execution of parallel paths.

[8] This would not require that the Intent documentation (or even the associated operator interface) map one to one into the Predictive Control implementation. Some form of compiler between Intent statement and implementation would provide any needed implementation discipline.

[9] But dealing in real time sequencing rather than computation.

- The execution of alternative paths.

The Figure 10 example with its text and embedded Sequential Function Chart show this. There are other sequencing forms as well as more general forms, which become important if one wants to cleanly integrate other forms of control into sequencing. The language includes:

- Looping.
- Continuous Operation.
- State Driven Operation.

For batch operation there is a NAMUR driven dream called a grundoperation which would surely meet the earlier criterion of preventing inappropriate usage. This is a sequenced element, which has predefined invocation prerequisites. A fully supported grundoperation would disallow its own initiation unless all required sequenced elements had already been executed and all required states reached.

ONE MORE CASE: A THREE LEVEL SMART SENSOR PROTOCOL

As a final example of the value of Intent Modeling: consider the problem of providing effective communication of Smart Sensor data, without multiplying the resulting data complexity. A Smart Sensor should provide the user with compensated measurement data,[10] but also provide notification of actual and incipient failure information in the most helpful way. The current tendency is to generate complex maintenance related failure data that just multiplies the information overload already provided to the operating people.

Consider the Smart Sensor by analogy with what would be expected from a smart person. Consider a field operator back with a set of obscure but operationally important symptoms about the state of a sensor (meaningful perhaps to the maintenance personnel). Suppose that operator rambled on without telling the board operator in simple terms how the sensor state affected the continued operation. Such a person would not be considered a smart or helpful person. So it is with the sensor. Smart Sensors have much more to offer if they make effective use of Intent Modeling based sensor validity coding.

This can be accomplished within a three level Validity Code Model:[6]

- Symptoms (Sensor Physics and Situation). The bottom level consists of the codes used to communicate precise device failure modes. The codes thus communicated have meaning only for the purposes of device maintenance.
- State (Device Utility). This is a set of sensor (or actuator) states and protocols reflecting the application useful state of the sensor in terms that are independent of device type. The device designer (or sometimes the application designer) would be responsible for defining the translation of the bottom level device specific codes into these more general application related States. There are at least two distinct categories of these States:
  - (Data) Validity. These States reflect on the validity of the data, its accuracy, repeatability, noise free character, etc.
  - (Sensor) Health. These States reflect on the continued viability of the device itself, independent of the current data validity.
- (Application) Role (Requirements). This represents a set of standard, general, application characterizations or roles. They allow the user to classify his application in general ways suitable for

[10] Perhaps with greater compensation for a wider range of circumstances than with a “dumb” sensor.

the comparison to the device utility States. In general, a Role defines which device States can be supported without changed control, and which changes in a control regime must be taken in the face of any change in Device State. Device failures would always initiate maintenance action. The Role includes two further categories:

- Function. This Role category defines the application requirements for Data Validity.
- Criticality. This Role category defines the application requirements for Sensor Health.

For example, Accurate data (a Validity State) is needed for Primary Control (a Function Role) functions, which affect product quality. But Repeatable data (a Validity State) is adequate for Secondary Control (a Function Role) functions, such as secondary cascaded controllers or feedforwards, which will be corrected by the affects of Primary Controls.

Similarly, Critical loops (a Criticality Role) cannot be operated at all without measurements; these require Unqualifiedly Healthy sensors (a Health State). A Smart Sensor validity code would include a matched set of these States and Role characterizations. These would permit the full automation of sensor failure responses designed to minimize unnecessary downtime or unacceptable crises. The automation would become effective once the application designer used the Application Role to define his application Intent.

CONCLUSIONS

Process Control is in a stage of awkward standardization, rushing to follow the digital world. This standardization can be premature if it misses important techniques, which would better capture our applications. Important to the advance of Process Control practice, as with any software, are practices that make our applications broadly readable without limiting their flexibility. Key to this is the ability to program application information models to transparently capture Intent. The concept of Intent is dependent on the application domain. Nothing can guarantee that engineers naturally document applications. But the absence of appropriate application languages will guarantee failure. Any coherent engineering practice lends itself to an underlying Intent structure. Surfacing this structure is a critical part to making that practice more generally teachable and accessible.

Bibliography

[1] E.H. Bristol, "Information Models for a Software Future We Never Know", ISA97, Anaheim CA, Oct. 5-9, ‘97.

[2] E.H. Bristol, "Deriving the Human Interface from the Automatic Controls", Automatic Control Conference, Philadelphia, June 24-26, `98.

[3] E.H. Bristol, "A Language for Integrated Process Control Application", Retirement Symposium in Honor of Prof. Ted. Williams, Purdue University, West Lafayette, IN, Dec. 5 - 6, `94.

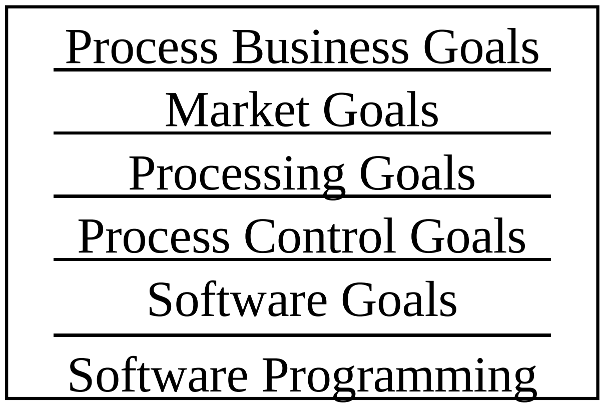
[4] E.H. Bristol, "Not a Batch Language; A Control Language", World Batch Forum, San Francisco, May `95; also ISA Transactions, Dec. `95.

[5] E.H. Bristol, "Redesigned State Logic for an Easier to Use Control Language", World Batch Forum, Toronto, May 13-15, `95.

[6] E.H. Bristol, "SuperVariable Process Data Definition", ISA SP50.4 Working Paper, Oct. 24, `90.

Figures

Figure 1. Levels and Parallel Support



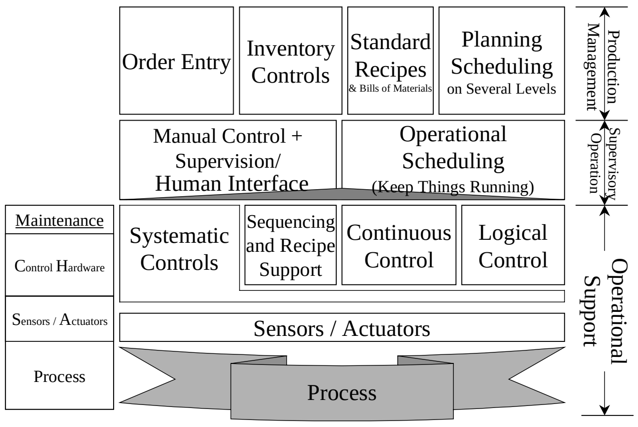


Figure 2. Physical Science Leveling

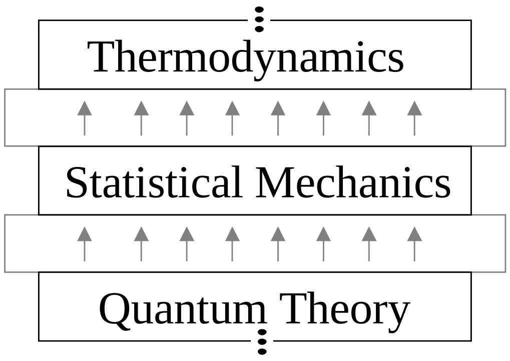


Figure 3. Subroutine Hierarchy

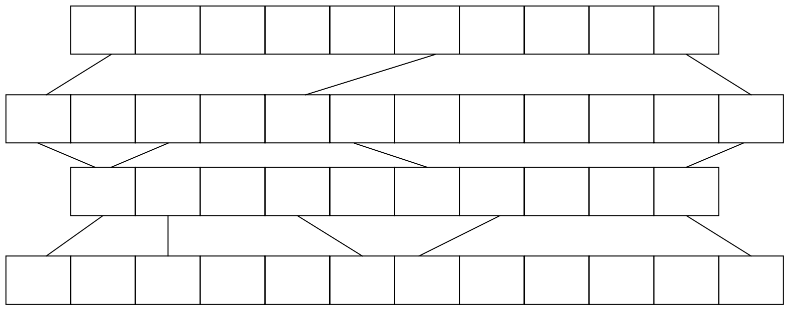


Figure 4. Sensible and Absurd Cascading

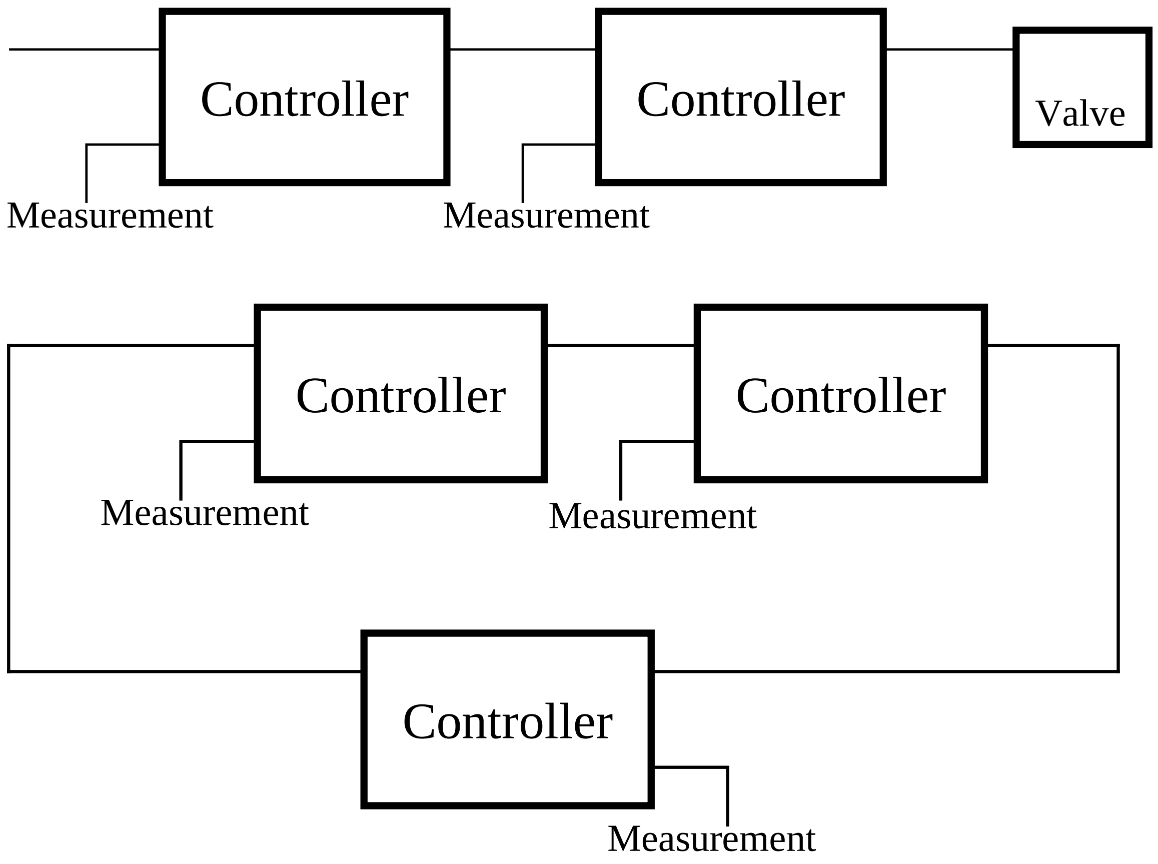


Figure 5. Algebraic Syntax

$$ax + by + cz = 0$$

$$y = ax^2 + bx + c$$

$$F = ma \qquad E = mc^2$$

$$\sin(2a)=2\sin(a)\cos(a)$$

$$D / F = (z_i - x_i) / (y_i - x_i)$$

Figure 6. C Code

```
while(isxdigit(command[i]))   {
  c = command[i++];
  if(c>=’0’ && c<=’9’) c-=’0’;
    else if(c>=’A’ && c<=’F’) c+=(10-’A’);
      else if(c>=’a’ && c<=’f’) c+=(10-’a’);
  bf = 16*bf+c;
  }
```

Figure 7. Complex Constraint-Switched Loop

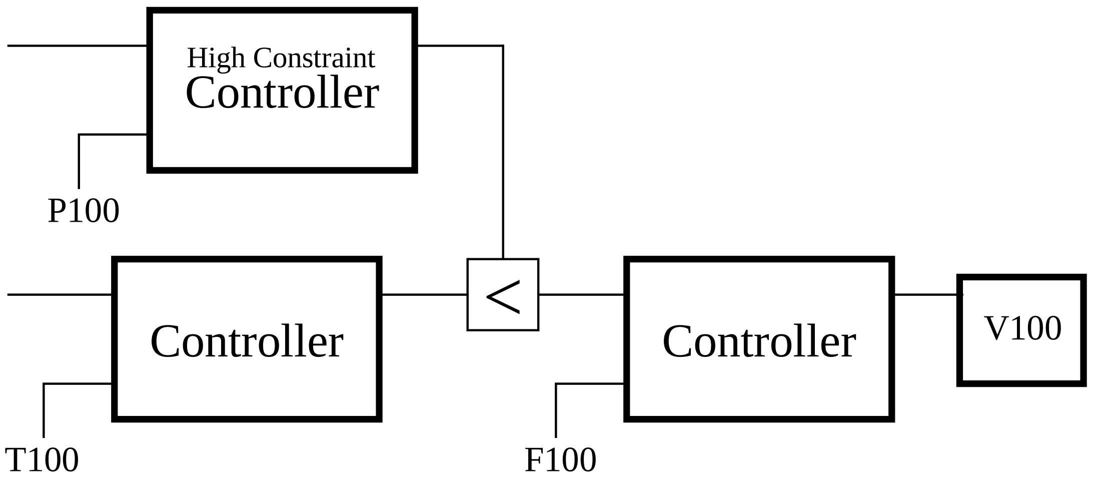


Figure 8. Automatically Generated Loop Display

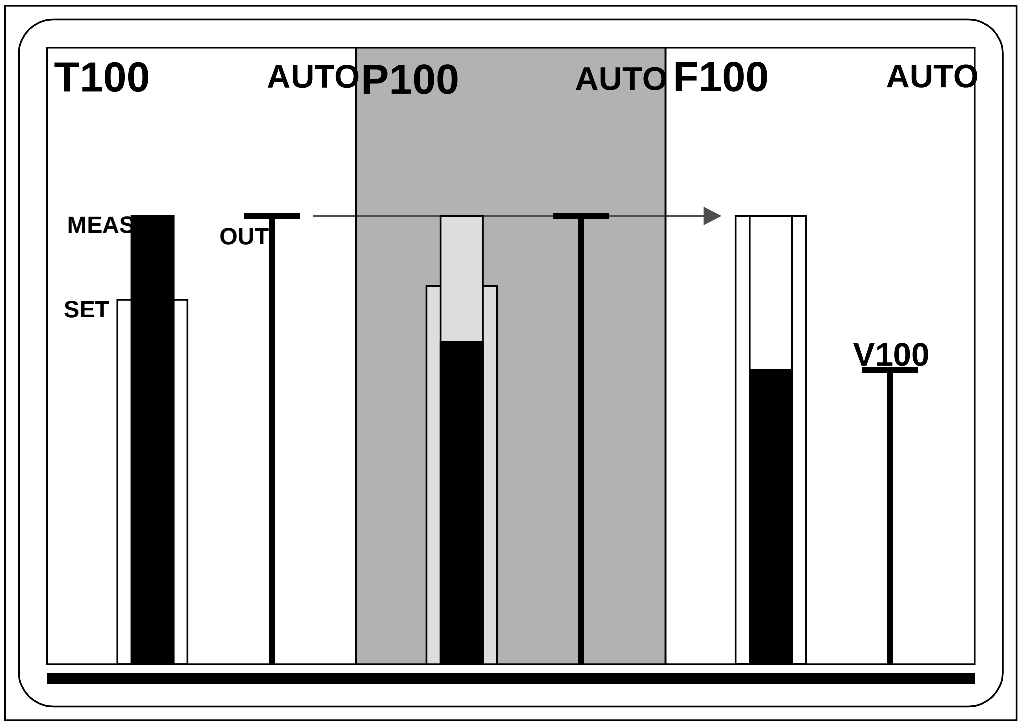


Figure 9. Fuel/Air Controls

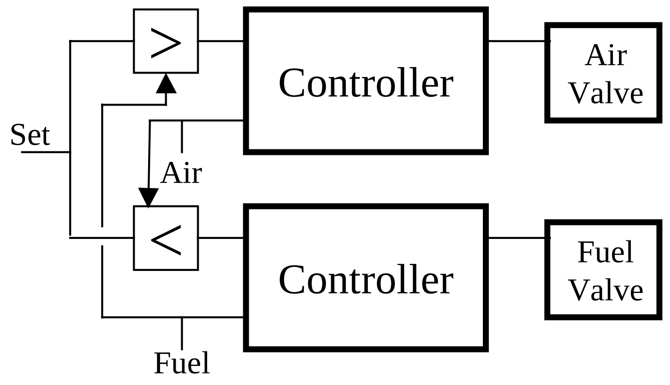
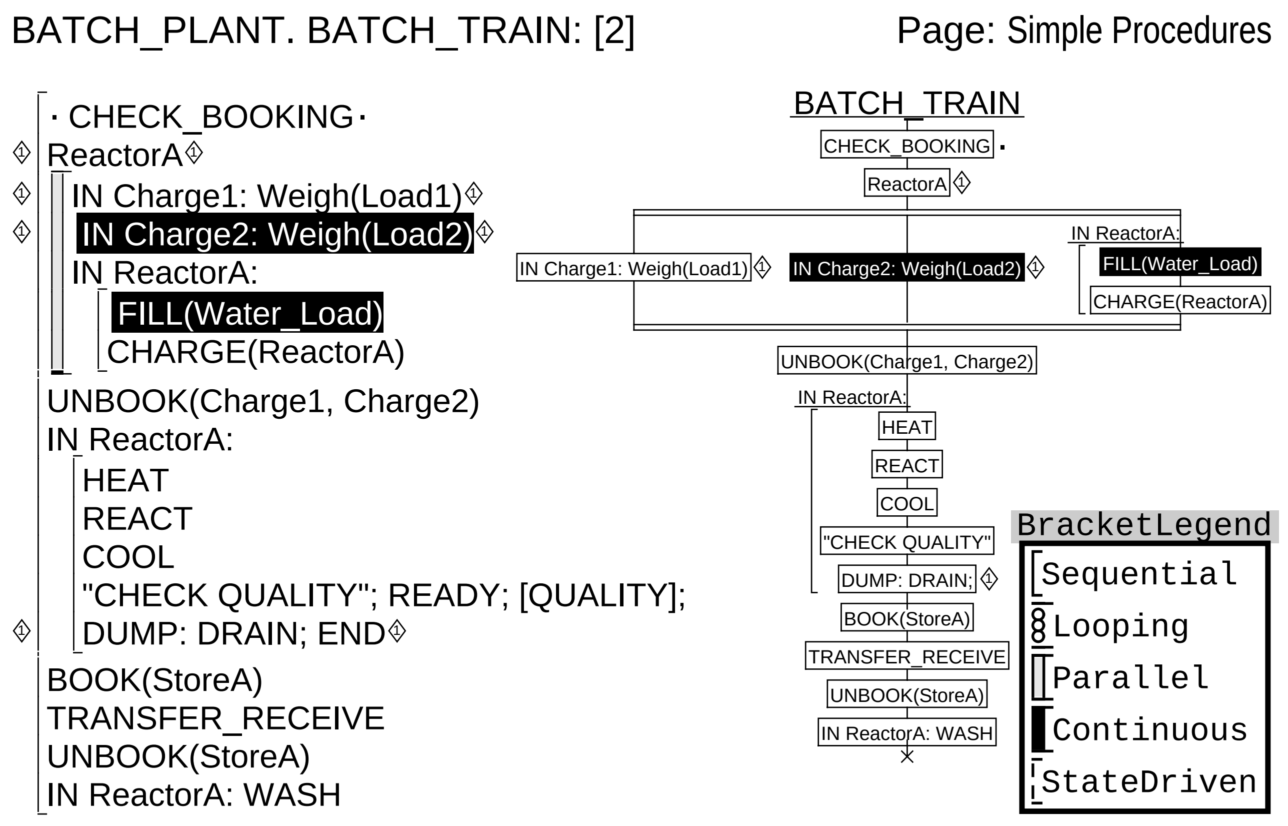


Figure 10. Semi-Graphic Text and SFC Sequencing



```
BATCH_PLANT. BATCH_TRAIN: [2]                    Page: Simple Procedures

· CHECK_BOOKING ·
ReactorA
  IN Charge1: Weigh(Load1)
  IN Charge2: Weigh(Load2)
  IN ReactorA:
     FILL(Water_Load)
     CHARGE(ReactorA)
UNBOOK(Charge1, Charge2)
IN ReactorA:
   HEAT
   REACT
   COOL
   "CHECK QUALITY"; READY; [QUALITY];
   DUMP: DRAIN; END
BOOK(StoreA)
TRANSFER_RECEIVE
UNBOOK(StoreA)
IN ReactorA: WASH
```