Logic control implementation from a process view

Ir Henk van Heun[1]

Abstract. State transition diagrams with additional timing can be used to represent the states of a physical process in a relaxed manner. These state diagrams are used to be the kernel of a logic control implementation. For a safe and reliable control it is not necessary to prove that the control program is correctly connected to its physical process at all times. In fact, when assuming that the information from the physical process is faulty or delayed, it is very likely some error condition will always arise.
This paper is about an implementation technique using the proper instruction order to prevent the huge numbers in combinatorial possibilities and staying close to the physical process when representing it. The resulting program is running on the highest speed possible and fully predictable.

Keywords: Logic control, hybrid control, state diagram implementation.

1 Introduction

History: In process automation some traditional fields of expertise exist. These fields of expertise show a parallel evolution with the development of the hardware components used to control a process.

Half a century ago the logic control was based on relays. Relays are working in parallel, so decision time depends on the slowest relay. The invention of the transistor led to the solid state logic of the first chips in the late sixties. Karnaugh diagrams, or the similar Quine-McCluskey [1,2] table based algorithm, were used to minimize the logic. The decision speed of this logic is related to the number of layers (=2).

The automobile industry, with General Motors in front, developed the PLC[2]. The logic control was brought into a programmable environment for flexibility in adapting the control when the process was changed. These early PLC programs were based on replacing the relay logic. Adding simple arithmetic and some analogue inputs and outputs was the next logical step. But handling the Boolean expressions seems to be unchanged. A PLC program is now a serial list of instructions and the decision speed of the complete program depends on the number of instructions.

The evolution in control logic programming is driven by the desire of creating reusable functions to enable quick development. Using “proven design” functions has

[1] Central Engineering Department NEM BV, Holland. Email: hvheun@nem-hengelo.nl. Industrial automation and software engineering.
[2] PLC= Programmable Logic Controller, an industrial computer to perform the control.

become the trend. Although these "standard" components can be correct working pieces of art, the interconnections are responsible for the (lack of) reliability of the total system. Time for a fresh approach I think.

1.1 Problem

"Standard" implementations are based on I/O (=Input/Output) related building blocks. Designing a logic control by connecting these multifunctional components leads to scattered process representations hidden in many Boolean expressions. Proving that the implementation is satisfying a functional specification is postponed until the commissioning phase or remains an unproven statement on paper.

Problem 1: In Fig. 1 shows such a "standard" building block expressed in a graphical manner:

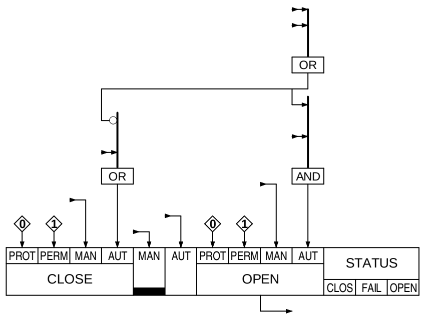


Fig. 1. Building block for a single valve.

This simple example shows the problem. A single Boolean output needs 10 Boolean parameter expressions. See appendix A for an explanation of this symbol.

The integration of manual and automatic commands is solved by this standard building block, but at the cost of multiple parameters. This block contains more than one memory (I counted 7!) and timers due to the strategy of responding to the last command and being able to survive transitional stages. Using this standard block, the operator interface is now a standard function also. Some "faceplate" can be shown on the screen and all alarms, signals and operator commands are neatly structured.

But when using these building blocks, the process is forgotten. The focus is on making flexible and reliable (single) outputs with a standard operator interface.

However it is normal to control multiple valves and pumps for a process. Like a "block and bleed" for a gas fuel system: three valves functionally tied together, no need for a single valve to be operated manually without operating the others. The standard building block is out of order here.

By using this standard approach, the decision speed and reliability of the total system is degrading. The standard building block solves some important issues, but creates problems also.

Problem 2: Another example of the problem is found in the normal standard sequence logic which can be implemented by using some SFC[3] language symbols in the PLC. Sequence logic is equivalent to premature state diagrams and can be used to express a Start/Stop-sequence by some organizing component and a sequence of steps. The next picture is copied from the ABB Procontrol manual[4] to document the Boolean behavior of a single step. Every DSC or larger PLC system is offering symbols with comparable functionality. This is an old and therefore simple example:

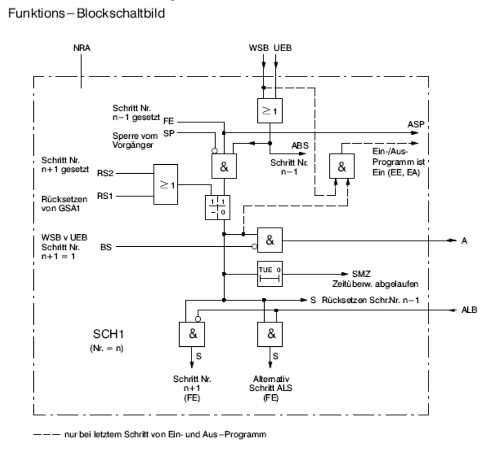

Funktions – Blockschaltbild
NRA
WSB UEB
≥1
Schritt Nr. n–1 gesetzt FE
Sperre vom Vorgänger SP
ASP
ABS
Schritt Nr. n–1
&
Ein-/Aus-Programm ist Ein (EE, EA)
Schritt Nr. n+1 gesetzt RS2
Rücksetzen von GSA1 RS1
≥1
&
WSB v UEB Schritt Nr. n+1 = 1 BS
&
A
TUE 0
SMZ Zeitüberw. abgelaufen
S Rücksetzen Schr.Nr. n–1
ALB
SCH1 (Nr. = n)
&
&
S
S
Schritt Nr. n+1 (FE)
Alternativ Schritt ALS (FE)
– – – nur bei letztem Schritt von Ein- und Aus-Programm

Fig. 2. ABB Procontrol example of a single “Step” function.

Some remarks about this logic to show the nature of the problem: Note that a Set-Reset memory is used to implement the active status of this “step”. When the total sequence exists of 9 steps, 9 memories are implemented. A total of 2^9 combinations emerge, so the surrounding Boolean expressions are needed to reduce that number to the functional 9 again. Note also that each step has an associated timer and only a single timer is active.

My biggest problem is not the overkill on instructions. But it is obvious that it will take at least 2 program cycles to make a proper transition (due to upstream and downstream connections). The designer of this logic was aware of that and combined the “state search” (UEB = Überlauf) with the prelude to a transition (WSB of the next step becomes True) to suppress the output signal A. (A is intended to give the AUT-Open command to the valve of Fig. 1.) During transitions the definition of the “step-active” is vague and therefore not suited to be used in combinatorial expressions. The only proper use is to connect them to the next step. It is easy to see that a single shot WSB (True for only one program cycle) can corrupt this logic.

When making Boolean expressions, the implicit assumption is always to combine variables representing their values at the same moment. Newtons law: (F=m.a) implies likewise that the mass and acceleration are valid at the same moment. It makes no sense to calculate the present force using the mass of last year and multiply it by the predicted acceleration of tomorrow.

[3] SFC=Sequential Flow Chart. One of the five IEC-1131-3 standard PLC languages. Like Graphcet.

[4] Druckschrift-Nr. D KWL 6311 96 D, Ausgabe 02/96. ABB Procontrol P manual.

The decision speed of a sequence program like this is at least two program cycles. This degrades the predictability. When proving the correctness of this type of program, all intermediate and implicit memories must be analyzed.

1.2 Solution

The solution is not complicated:

- The process representation should be made explicit and transparent.
- The intrinsic implementation technique of the logic control should be proven correct to obtain reliability and predictability.

Philosophical remark: When proving the correctness of a system, the physical process and its control are normally seen as a single system. For logic control I disagree to do so, because the transparency of the process representation becomes cloudy. After reading “Laws of Form” [3] from George Spencer Brown[5], I realized that making a distinction between the physical process and its representation has deep consequences. The representation only highlights and extracts some symbols from reality. The representation is therefore not strictly connected to the reality. And as Spencer Brown argued: when making a distinction, you are dealing with three objects: Two distinct areas and a border.

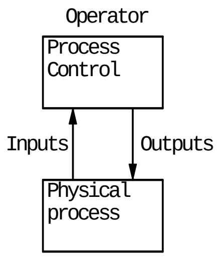


Fig. 3. System decomposition: Operator + Process Control + Physical process

Using techniques from the Operational Research to optimize a system does not address this fundamental issue. When a system is controlled, the physical process and its control are separate worlds, connected through some border. This border will transfer only a selection of information and introduce delays and errors. Control logic design should handle this. Fig. 3 shows the separate worlds. The Inputs and Outputs form the border and you could include an operator on top also.

Designing a control: When building a logic control, the key question must be:

How do we represent the physical process?

The most important task of the logic control is to keep track with that physical process. Of course the Inputs are indicating the process state, but Inputs are not always error free. You need to decide when an Input can be used and when it must be ignored. If the process is represented by a state diagram, the actual state (not the

[5] He was familiar with control logic. He even holds a patent for a Lift Control System in 1963.

process itself, but its representation) is the perfect means to define the usefulness of an Input. When using the process view as a fundamental approach, the state transition diagram comes naturally. How I define (and implement) such state transition diagram will be explained in great detail in this paper.

The answer to the key question has two components: the representation itself (the states) and how it keeps track in time (transitions and timing). My implementation technique deals with both.

Nasty property propagation: The state diagrams are a direct representation of the processes to be controlled. The Inputs are used in the transition expressions to establish the actual state and the Outputs are based on the states.

Outputs are therefore *no direct functions* of Inputs!

When building Boolean expressions for Outputs using some memories and Inputs mixed with commands, all nasty properties of faulty Inputs will be transferred to the Outputs immediately. The state diagram absorbs the nasty properties and the Outputs solely based on the states are error free again. The proper synchronization with the physical process is the first and only task for the state diagram implementation.

In your body an immune system is effectively active at all borders. All attacks of foreign bodies, all nasty properties of the food are dealt with. Its goal is to isolate the problems. And after this effective filter your healthy body can react in the real world according to its functionality. Your muscles react directly on the commands of your brain, no checking of the fuel or doubting the commands found here.

The implementation technique of state diagrams is very important, the reliability depends on it. When the implementation is pushed away into some “standard” components, the basic condition for a reliable control is absent. Randomly joining components is not a good engineering practice. Understanding the nature of a process, and adjusting the implementation accordingly, is.

Implementing the solution: The next chapter shows how a single execution of the serial list of instructions takes all decisions. The implementation is based on clustering the instructions, using global data and a redundant representation. The resulting *single cycle decision principle* creates reliable and predictable logic control.

Clustering instructions into categories and being aware of the cluster order is one of the applied rules. Appendix B explains the clustering details.

In this paper the solution is demonstrated with an example using my HCADwin tool. Appendix C explains some background of this soft PLC and simulation program. The example implementation is described in great detail in appendix D to demonstrate the implementation technique.

Timing: The “relaxed or elastic synchronization” between state diagram and the physical process is a given fact. Inputs can be delayed or faulty, so it is a reality that the control logic will be misinformed at some moment. Here the timing implementation (using a single timer) of the state diagram comes into action. It only matters that the order of states is predictable and that synchronizing is realized in the long run. Starvation (=infinite waiting for some condition we know that will never happen if some Input fails) is solved by the trip time parameter. The timing function sets a single ESD Boolean which can be used to “trip” the process.

A state diagram should have one or more “trip” states when the corresponding physical process is to be controlled and directed to these safe positions.

These trips[6] are the ultimate means to synchronize the physical process and the state diagram again. It could be done by some “reset” command from an operator also. And the trips provide a solution for “deadlock” and “starvation” situations caused by unexpected failures of the Inputs. Consider the trip logic as an additional safety barrier. It is good engineering practice to create multiple barriers. Creating fully integrated and optimal solutions is often equivalent to creating problems.

Making a state diagram using this timing feature allows you to minimize the number of disjunctive states and keeping the transition expressions simple as well. No thoughts have to be spent on complex combinatorial situations. Assigning and defining the Outputs is purely based on ORing the relevant states. By definition this state context is correct.

The timing parameters for each state can be optionally used (see appendix E). The combination of Inputs, ignored in all but relevant states, and the state diagram timing offers the transitions to the trip states for ultimate synchronization.

1.3 Safety aspects

When creating complex systems, one should at least think about the correctness. Model checking is one way of discovering flaws in the design. But in real systems it does not always matter if some, minor or major flaw exists. Availability, reliability and predictability are more important.

> When living in the woods and the TV shows a nearby fire, you will hate your car when it refuses to start because of some failing brake light. You would even accept a square wheel if it would lead you out of the danger zone. The correct deduction that the “car system” is not healthy is not always that important.

A control system based on independent working parts and ignoring or dealing with faulty information can be very reliable and in itself proven correct.

Realize that only two categories of errors exist: *known and unknown*. The latter can not be blamed to be forgotten by an implementer. I try to treat the first category alike and ignore them also. But the attitude is to incorporate all deviations naturally.

Functional specification: In the industry the functional specification is often expressed in “narratives”, short stories expressing the purpose of a process, how to start and stop it and how to deal with some deviations. The safety aspect can be specified in a “cause and effect matrix”, linking certain process conditions to forced output patterns. This is the design perspective on a process, ignoring the complexity and flexibility from a more dynamic perspective. These functional specifications will not lead automatically to a proper implementation of the total control.

Using state diagrams with trip states is combining all aspects of the functional specification in a transparent fashion.

[6] Trip is the industrial term for the ESD (Emergency Shut Down) procedure for reaching a safe state in the quickest way possible.

Control correctness: I assume that the machine (like a PLC) executing the control is working perfectly. It executes my serial list of instructions without any errors. When this machine is expected to fail, some additional hardware is added to assure safety. Like safety valves to be opened by a high pressure of the medium, or forced valve positions at power failure conditions. The logic control is implemented solely by means of state diagrams. For hybrid control the state diagrams are determining the context for the analogue controls. The control is correct when all states are checked.

Model checking theories like the *timed automata* [4] often assume a single system. By defining a complex system as the product of component systems, every system can be seen as a single system. The physical process and its control are joined together in a single system as well. I reject this mix due to the faulty interface of inputs and the infinite number of events in the physical process. At least the control and the physical process must be separately dealt with for ignoring irrelevant events. For model checking and proving phenomena like deadlock or starvation, this theory is well equipped. It is an analyzing tool, but not suited to create a control.

2 State diagram implementation

This paper defines a method to create a control based on two important issues. The first is the process representation within a controller and the second is the set of implementation rules to obtain reliability and predictability.

The standardization of today forces designers of control programs to use heavy weight standard building blocks focused on handling the I/O (Input/Output) like the example in Fig. 1. These blocks are "structured" for the general cases, forcing the programmer to invent Boolean expressions to enable/disable the options of these standards. Within these Boolean expressions the process representation is hidden. Why not start with the process perspective and express it transparently and directly?

Technicians are tempted to (re)use proven building blocks. And for a good reason too. Sometimes the real problem can be addressed by standard components, but connecting these components can be complex and introduce timing issues.

Process representation and implementation rules: Creating a (hybrid) control is finding the proper order of the serial list of instructions (implementing), based on the correct representation of the physical process (global administration). The implementation should avoid all additional memories to maximize the predictability and minimize the combinatorial space.

> Some philosopher once said that all systems are imperfect when memories are involved. I could not agree more.

This chapter contains the receipt of implementing the proper order in the serial list of instructions. Without the proper order the decision speed of the total program will become a multiple of the cycle time and therefore introducing unwanted timing problems and obscure memories. When decisions take multiple cycles, they are based on a mix of actual and historical data, which is bad. Inspection of the instructions will not reveal that quickly.

The exact order of the instructions is not always important, but clustering the instructions into functional entities with respect to the state diagrams and keeping the cluster order intact is a key issue.

Designing: The first step when implementing, is to understand the nature of the physical process. Each parallel physical process (PP), important to represent, is given a name (like PROC). Within the PLC it is a global variable of that name holding an integer. This variable is called the State Variable (SV). Every process state is numbered.

This is representing the process: physical states are represented by a number. Assigning zero to the initial state when the system is in hibernate state, is good practice. Using a limited number of states is good practice too. If over 99 states are needed, some suspicion of bad design comes into mind. A state diagram should fit on a single page. This compactness is not a real demand, but when informing a second or third party it is crucial that it can be understood effortless. When shown on a screen for an operator, he should see the complete diagram in a single window.

A trivial, but always forgotten property of a variable holding a number, is the impossibility to hold multiple numbers at the same moment. States are expected to be disjunctive so this “single number” property of the SV is used. By its very nature not a single instruction is needed to remove the previous number. Implementations based on separate state Booleans always need some complementary set of instructions for resetting old states. This corrupts the “single cycle decision taking”, causing delays or mysterious errors.

My implementation technique never creates wrong “reset” logic: *it is absent*.
The next step in creating the control logic is implementing the clusters of instructions. The ultimate goal of these instructions is to keep track of the physical process. The SV should reflect the state of the physical process. But it is not important that PP equals SV at all times! In a relaxed manner the representation of the process should follow or precede the physical process. For a reliable control it is more important to know that each program cycle only one state transition can be made to fulfill the internal criteria than knowing “PP equals SV”.

Now we define the implementation clusters in their relative order, needed to implement the state transition diagrams. Keep the remarks of the clustering details in appendix B in mind:

2.1 Preparation cluster

The first cluster of instructions in the serial list is creating some Boolean variables to be used as transitions in the next clusters. Typically the Inputs are used in these straightforward Boolean expressions. Like comparing a pressure against some constant, or reading the status of a switch. Knowing that a transition is used to leave specific states will simplify the expression. The value of the expression is irrelevant in all states not using that transition, use that freedom to keep it simple.

2.2 Trip cluster (optional)

This cluster of instructions is creating the trip transitions. This is a special category of transition Booleans. Typically some Input is combined with an OR of the states in which the trip transition can occur. By connecting an Input like “Pressure too Low” with states like “Almost active” or “Active” the resulting “Trip on low pressure” signal can be used as an alarm at any moment. This signal will be used solely or in combination with other trip signals as a transition to a trip state.

The naked “Pressure too Low” Input is likely to be a very normal signal when the system is “Sleeping”. Although the Input description contains “too Low”, it is *not true* unless seen from the “system running” perspective. This is a common mistake for many designers. Descriptions are context sensitive. My trip signals are a combination of the naked signal and the proper context and therefore real alarm signals: only True in the proper context.

2.3 First failure cluster (optional)

This cluster uses the trip Booleans and normal Inputs to distinguish the first failure. Although I am aware that the Input sampling mechanism of the PLC gives no accurate insight in the exact timing of the events, it is good enough (and the best available) for indicating the reason why a state diagram started some emergency procedure. From the state diagram perspective this First Failure was the precise reason to go to some trip state. It is important to ignore all other alarms and signals, to prevent the operator being overloaded with unnecessary information.

A First failure alarm group is implemented as an integer variable. When zero, no failures are present and the very first failure writes its number into this variable. All other failures are ignored if the FirstFailure is non zero. A deliberate reset command (from an operator or a state diagram) writes a zero to activate the next FirstFailure capture.

The First Failure signals are often identical with the trip signals. For that reason the Trip cluster precedes this cluster.

2.4 Transition cluster

This cluster contains the transition logic. Typically a new number is conditionally written into the state variable (SV).

```
{Pascal example} If PROC3 And Transition5 Then PROC:=4;
```

The exception is the trip Boolean, which writes a new number without a state Boolean involved (but trip Booleans included an OR of state Booleans themselves, so the structure is the same).

Note that SV (PROC) can change more than once in this cluster. But using the state Booleans (PROC3) in the conditional part assures me that the last SV update is valid and only a single transition is being executed.

2.5 Redundant representation cluster

This cluster is creating the state Booleans by testing the state variable:

```
{In Pascal syntaxe} PROC3:=PROC = 3;
```

It is not allowed to write the state variable PROC outside the transition cluster nor to write the state Booleans PROC#[7] outside this redundant representation cluster.

The state is now represented in two different ways. Primarily by the state variable, but for easy usage also as a set of disjunctive true Booleans. The programming order of the clusters and the "single number property of the SV" guarantees that. Also important: the order of instructions within both clusters is irrelevant! This is important when proving the correctness of this approach. No cases involved, no permutations of instructions to be checked, just the proper cluster order taken into account.

Normally redundancy can cause problems. Which information to believe? Is there consistency of both representations? The cluster order solves that problem. The representation of the actual state using an integer creates the disjunctive property and is very efficient for an operator interface to show the status. The representation of the state Booleans is very convenient in the use of expressions. Every state Boolean can carry the description of the state for documentation purposes.

It is not a programming trick, but a means to an end: I want to know for sure that the mechanism is *fast, reliable and easy to document*.

2.6 Timing cluster

This cluster is using the fresh state Booleans to copy the time parameters of the actual state into three global variables. After actualizing the time parameters, the elapsed time PROC_T and three comparisons can be made. Each state diagram has now three timing Booleans assigned. The "Alarm Low" (PROC_AL) Boolean is used in the transition cluster to ignore the transition Boolean when a state is recently active. The "Alarm High" (PROC_AH) is used for generating a "too long in step" alarm to alert an operator. In some applications the associated outputs could be suppressed on this Boolean also. The "Emergency Shut Down" (PROC_ESD) is used in the trip cluster. Some states are time limited by these trips. Infinite waiting for some failing process condition must be avoided. This ESD Boolean implements that elegantly.

See appendix E for the program code example.

2.7 Output cluster

This cluster is using the state Booleans (with an OR) in expressions to determine the outputs. One can think that the output pattern is completely determined by the value of the SV. In some cases an output can depend on multiple state variables from parallel processes, but it is good practice to assign an individual Output to a single

[7] # stands for the state number. The state Boolean is prefixed by the SV name.

SV. Operators will notice the relation between the SV and the outputs and understand that in important terms of predictability.

3 Implementation consequences

By repeating the program cycle, these clusters of instructions are executed over and over again. Notice that by its very nature the SV can follow only a single transition. Multiple transitions can not be executed in a single cycle. This corresponds with the process itself, which is assumed not to be able to jump to any other state. When a process is able to “jump”, the state diagram reflects that.

When two or more state diagrams are implemented, it is important that the order of clusters for each state diagram is correct. No compromises within a single state diagram are allowed. The clusters of different state variables may be intertwined, but normally it is good practice to order 1-1, 2-2, 3-3. (Functional clusters together). If the state diagrams show some top-down ordering, I tend to program the top diagrams first and down diagrams later, because it is the natural flow from global decision to commanding the output. The down diagram can use top diagram state Booleans as transitions. Notice that the handshake from down to top will take an extra cycle.

This approach has some important features:

Decision speed: If the cluster order is correct, every diagram uses the latest inputs to make a transition or not. All outputs are based on the most recent states. The outputs are therefore always based on the most recent available data. Every program cycle all decisions are made. Not a single decision takes more than one cycle.
For comparison: No SFC implementations I know of can guarantee to “step” in a single cycle due to a wrong (read single) representation of the status. See Problem 2. For the reliability of a control it is very important to know this for a proven fact.

Because of the cluster order, it is impossible to have more than one transition in a diagram in a single cycle. When using an OR of state Booleans to create an Output, you are guaranteed that every cycle this OR will remain TRUE regardless of transitions made when the collection of ORed states was never left. Also a single shot state (visited for only one cycle) can be guaranteed to be seen TRUE in any cluster if the diagram passes that state. (My equivalent of knowing that all “events” are handled) Due to these properties elegant programs without additional memories or signal prolonging timers to deal with multi cycle decisions, can be implemented.
What if there would be the need to skip a state? My first remark would be that is it pointless to make more decisions than one in each cycle because the outputs can not be seen by the physical process until at the end of the cycle. If the nature of physical process asks for it, than an additional transition to “jump” to another state is added of course. No objections to create a full graph for a state diagram if that is considered meaningful. (It shows bad design however: ever seen an elevator skipping a floor?).

Handshaking between two state diagrams: Normally the decomposition in parallel processes does not include the complete autonomy of each process. Consider an elevator process. At a certain floor the elevator is waiting for someone to enter. This "person process" was autonomous until pressing a button to call the elevator. Both processes are synchronized for the duration of the travel to another floor. The corresponding state diagrams share the same characteristic. Both can have a "wait" state for each other.

Easy to implement, because the state Boolean of one diagram is a transition Boolean for the other. When a mirror construction is used, both diagrams can be forced to wait for each other and the joined effort can be guaranteed. When using the timing as well, the elevator will not wait forever and assumes some faulty input if the person walks away. It just opens and closes the door after some time. Without another request it could wait here, or on some other level.

3.1 State transition diagram semantics

Like in model checking, and in various other fields, some form of state transition diagrams is used. My state definition is slightly different. Before complaining about it, realize that semantics are important. George Lakoff [5,6] showed the power of the metaphors to convey knowledge. The terms I use can be best understood by having the implementation technique in mind and studying the example.

First of all a state is an abstract process state, implemented by a natural (integer) number. The global variable holding this number is called the state variable (SV), but for all used numbers within the same diagram a state Boolean is used also (e.g. SV3, meaning "SV was 3" in the first four clusters and meaning "SV is 3" after the fifth cluster (redundant representation). Note that the subtle shift of meaning in the fifth cluster expresses the awareness of validity in time). By following the order rules (clustering the categories) these Booleans are guaranteed disjunctive in all but the creating cluster (when SV on turn is unchanged). Both state representations are representing the state at all times outside their manipulation clusters.

This double representation assures the important property of making no or a complete single transition every program cycle!

And when using or reading, the state Booleans are convenient and easy in any documentation[8]. Without some double representation, it can be proven that implementation in a serial list of instructions is impossible or imperfect (like multiple transitions/cycle or multiple cycles/decision).

The naming convention for the state variable (PROC) and a single state Boolean (PROC3) helps reading the diagrams. Don't underestimate the importance of the proper documentation of the state Booleans. It is obliged to achieve reliable programming.

Well formed diagrams: A complete diagram has two or more states, connected with directed arrows, the transitions. Each transition is effectively a Boolean variable,

[8] Documenting the State Booleans is documenting all valid SV numbers.

not an expression. If an expression is needed, some Boolean expression can be made in a preceding cluster. In my drawing tool, HCADwin, the transition Booleans can be used normal or inverted. When the timing is used also, the elapsed time for the actual state can be added using an AND, giving four ways of using a transition.

The diagram must be well formed: each state must be reachable from every other state (multiple steps allowed), assuming that the transitions are cooperative. A designer will look into that and omit silly states or add transitions. When dynamically states are unreachable due to false transitions, it looks like a dead end, but using the trip transitions it is not. Although I dislike it and tend to forbid it, it is possible to use a transition from no state to activate a state. Typically these transitions are used to force an initial state for the very first cycle of the program. Or a single shot operator command to reset some diagram. In my view the creator of the diagram using these *deus ex machina*-transitions is lazy and undermining the correctness.

3.2 Dancing connection

The primary goal of my process control is keeping the connection of the physical state and its representation. Knowing that inputs are faulty or delayed, it is impossible to have a 100% strict connection. But is that necessary? No. When a diagram follows the physical process to a certain extend, it is good enough. When exceptions arise, some special transitions can be used for starting a path in the diagram for exception handling. A flexible, almost dancing approach, is all we need.

The connection between real world and state diagram has 5 cases:

1. *Actual:* both states correspond. Ideal situation.
2. *Future:* diagram shows future state. This is normal when some output is changed and the process needs time to react. Like opening a valve.
3. *Past:* diagram shows past state. Again very normal when some delayed input is telling us the status. The elevator reaches a floor before the input used is telling us so.
4. *Wrong:* the diagram is disconnected. This is fun. If this happens and the diagram is well designed, it is a matter of time to synchronize again. With “well designed” I mean “self repairing” by using the additional timing or alternate inputs. At first the alarm timing could be used for an alternate path (normal Input was faulty). At last the trip transitions are used to enter a safe state, forcing the physical process to sleep.
5. *Absent:* SV contains a number without an associated state Boolean. All state Booleans are false, so no transitions can ever be made. This absent state is forced by me when starting the commissioning of the program. Like entering state 99 for the elevator position. All associated outputs should be in a safe position now (hibernate state). If the physical process was acting, it should come to a standstill. Now look at the real elevator position. Is it on the first floor, corresponding with “SV=1”? Just enter the proper number and your program is picking up at the correct point.

The “wrong” and “absent” cases need not to be investigated. This is very important for understanding my philosophy. In fact I claim that the representation of the physical process is consistent when all diagrams are well formed. It is much easier to prove that this representation is correct in itself than dealing with nasty properties from the real physical process. When accepting that the physical process does not exist in terms of polluting the representation with undesired events or properties, the “wrong” case is non existing. This is a very pleasant product of assuming the poor quality of the inputs. (And Plato, see epilogue.) This is not the only consequence of turning the world upside down. My state diagrams always ignore all inputs unless a transition expression is using it for leaving the actual state.

When modeling a system, one must be careful. The state space is easily exploding to enormous proportions. A lot to investigate and unfortunately: it all must be right.

John Grey [7] showed how utopia-thinking has caused human disasters. Realistic thinking is better.

System philosophy: The real time systems I programmed were all based on the following principle: Start with a sleeping system. This state with minimal action can be proven to be stable, stationary and correct. All representing state diagrams are in the initial state. All associated outputs are off or whatever is the safest condition. The system is “proven correct”. (At least is should be easy to prove)

Wait for the “wake up call”. Some Input or operator command will start some activity: “Open one eye and check the daylight”. The “wake up call” transition includes the lack of error (by definition or assumption, because it is all you can know about the physical process), at least at that moment in that state.

Now I know for sure that the program is a little active and still correct because of the (assumed) error free status. It remains in a “proven correct” state. (If still dark or ill, don’t get out of bed.) This awaking continues until we reach the utmost alert and active states. When some unexpected inputs could cause some trouble, fall asleep again. (If going to your work and the train is gone, try your bike. If the bike is stolen, go back to your bed and retry tomorrow)

This “full induction” scheme is proven correct in all visited states. All we need to know is the correctness of the transitions when stepping and the willingness to fall asleep when the inputs are failing or show strange behavior.

High availability: It is good practice to ignore a single fault. For example in the GLT project[9] this strategy was used. Assuming that all inputs are correct, the availability would be 90%. Accepting a single error increased that number to 99.8%. So my “return to sleep” on unexpected signals would make a very safe system, but only 90% available. Implementing some alternative routes in the diagrams or using parallel transitions to cope with single errors would be still safe enough and perform much better. This is a good way of dealing with faulty information and easy to incorporate within the state transition diagrams.

A strict mathematical analysis using all available information at all moments could describe a system which is optimal but shows a poor availability.

Ignoring irrelevant data is the key to safe, reliable and high available controls.

[9] GLT=Groningen Long Term. Dutch 2 billion EUR project prolonging gas field usage.

3.3 Example implementation

Appendix D shows an example of the state diagram implementation using my HCADwin tool. All implementation rules are strictly obeyed.

When using a graphical drawing tool for a state diagram, it is impossible to generate a cluster of instructions for a state symbol and follow my implementation rules. Every HCADwin state symbol generates two sets of instructions. The first set is placed in the transition cluster, the second set is “delayed” to be placed in the redundant representation cluster. In a graphical environment the “page” is a natural entity. Therefore the “page end” is the proper position for the delayed instructions, unless to be forced earlier by a “Flush” symbol to enable a single page holding both clusters (and the timing cluster as well). See Fig. 6.

Now I can guarantee the single cycle decision. When code generation is based on a cluster of instructions for each symbol, which is very normal, then my important principle for predictability is violated.

4 Epilogue

Plato’s *idea* of a circle is found in realistic circles. But a realistic circle is only visible when drawn with a certain thickness. The charm of the *idea* is the pureness and therefore flawlessness of the concept. The state diagrams are representing the physical process with this pureness and flawlessness.

The obliged clustering of instructions of the program and the double representation are important for the guaranteed single cycle decisions and the 100% reliable use of the states in the rest of the program. By its very nature this kind of program is easy to develop and even easier to commission. The standard focus on the actual state makes it possible to change a transition or add some state or alter the output pattern without any interference with other states. What you see is what you get, transparently.

Using the timing and trips to restore synchronization is the “self repairing” property of my control. Like the weak chemical bond of the DNA molecule makes it easy to destroy. But being the most likely connection at the normal temperature, it restores itself in the old configuration. This weakness in combination with most probable configuration, made the DNA structure exist for millions of years.

The tendency to return to the initial or sleep state seems to contradict a system designed to be available at all times. In my experience the utmost alert state will be manifest almost all of the time. And if some “wrong” state occurs due to dirt, faulty inputs or old age, my system resumes much faster. Systems designed to keep working are magnifying the problems and working a little longer, but causing more damage and therefore increasing the down time.

Summary: The developments in software engineering are troublesome for the modern DCS. The marketing techniques are shouting how easy it is to connect the standard components and “fill in the dots”. In every release of the standard components, the amount of nice looking functions increases. Therefore the amount of

Boolean expressions connected to the parameters is growing and the transparency is decreasing.

My tobacco transport program was based on my approach and was running for 17 years without any alteration. My first program for a Teflon factory survived the migration to new hardware and is still running for the last 25 years. The state diagrams were completely captured in data to enable recipe switching. It was implemented on a higher abstraction level. Downloading new data before a Teflon batch, altered everything except my cluster order and the data interpretation program. The key issue is how the physical process is represented in a relaxed manner.

Acknowledgments. I would like to thank Johan Schmaal for his contribution of integrating the First Failure within the state diagrams. Fig. 12 was created with his tool. And I would like to thank Rom Langerak for encouraging me to write this paper from the "caves of industry".

5 References

1. McCluskey, E.J.: Minimization of Boolean functions, The Bell System Technical Journal,1956
2. Quine, W.V.: A Way to Simplify Truth Functions, The American Mathematical Monthly Vol. 62, Nr. 9, 1955
3. Brown, S., Laws of form, London, Allen & Unwin, 1969
4. Alur, R.: Timed automata NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems. A revised and shorter version appears in 11th International Conference on Computer-Aided Verification, LNCS 1633, pp. 8-22, Springer-Verlag, 1999.
5. Lakoff, G., Johnson, M.: Metaphores we live by: University of Chicago Press 2003
6. Lakoff G., Nunez, R.: Where mathematics come from; Basic Books 2000: ISBN: 0-465-03771-2
7. Grey, J.: Black Mass, ISBN 0-713-99915-2; Allen Lane 2007
8. Wirth, N: The Programming Language Pascal. 35-63, Acta Informatica, Volume 1, 1971.
9. Knuth, D.: Sorting and Searching; Addison-Wesley Professional 1998: ISBN 0-201-89685-0
10. Henzinger, T.A., Wong-Toi, H.: Using HYTECH to Synthesize Control Parameters for a Steam Boiler In Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, Lecture Notes in Computer Science 1165, Springer-Verlag, 1996, pp. 265-282.
11. Stephanopuolos, G.: Chemical Process Control: An Introduction to Theory and Practice; ISBN: 0-13-128596-3; Prentice-Hall, 1984; Page 634.

Appendix A: Explanation of Fig 1.

The large rectangle stands for a standard building block (a “typical”) containing some logic instructions, memories and timers. The block is to be connected to a digital output of the PLC. The figure describes in great detail how this block is dealing with the manual commands from the operator (the MAN connections on both CLOSE and OPEN sides) or from some automatic procedure (AUT connections). The MAN + AUT connections in the middle are (re)setting a status memory for selecting which captain (MAN or AUT) on the ship must be obeyed. The PERM connections are close and open permissive conditions. A permissive condition prevents the operator of giving a wrong command by checking an important process status. A permissive is used to determine the proper context for a transition, but is not flawless due to the use of Inputs.

When the process enters a state that the valve must be closed (or opened) to prevent serious damage, the protection commands (PROT) are used. These protection commands will overrule all operator and automatic commands.

Notice that a programmer must invent 10 Boolean expressions to fulfill all parameters needed for this standard block. And this is even a simple example! When analyzing these parameter expressions, you will see that the upper right OR is the desired valve status coming from some normal control logic (details are left out here). The additional AND is an extra permissive for the automatic OPEN command. The other OR gives some refinement for the close command. (An inverted connection is indicated by a small circle.) All lines with an arrow on the left side are connected to some input or other (complex) Boolean expression. The OPEN result underneath is connected to the PLC output.

Appendix B: Clustering details

For the instructions examples I use the Pascal [8] language. It is an elegant language. The PLC is executing a serial list of instructions. Let us investigate what that means.

First of all is the order of instructions always the same. And all instructions are executed at every cycle. The control logic is implemented by this repeated list of instructions. Consider a small program:

```
{Program 1} A:=not A;
```

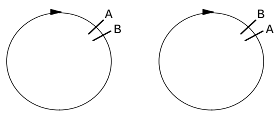
This is an oscillator when repeatedly executed. The frequency is directly related to the cycle time. A complete period takes two cycles. Now consider:

```
{Program 2} B:=A; A:=not B;
```

It is the same program, but it takes two instructions. And what about:

```
{Program 3} A:=not B; B:=A;
```

I changed the order of instructions, but because of the cyclic execution of the instructions, the result is the same. Is it really?



Instruction order and clustering.

The above figure illustrates that for these two instructions the relative order is insignificant: A is followed by B in both cases due to the repetition of all instructions.

The instructions are put close together, *they are clustered*. Observe how the global variables A and B are changing solely within the cluster. Outside the cluster you will see a difference between program 2 and 3. The value of the expression `A and B` differs. Program 2: always False, Program 3: always True.

This is the source of many errors or unreliable results. You should be aware of the order of instructions and the relative position within the cycle when you are using the stored variables. Another trivial example:

```
{Program 4} C:=A; B:=C; A:=not B;
```

Is functional equivalent to program 1. But now I can really change the order into:

```
{Program 5} B:=C; C:=A; A:=not B;
```

And the behavior is changed at last. The oscillating frequency has changed. It takes 4 cycles for a complete period. Using a memory to delay (or better: postpone) can be useful. In your Boolean expression however it is not immediately clear that the used variables are recent or ancient history. When the order of instructions is neglected or forgotten, some unwanted memory is introduced. The result of clustered instructions depends on the order of the individual instructions.

When constructing some “subroutines” for convenience, the same argument is valid again on a higher level: the result of some clustered subroutines depends on the execution order of the individual subroutines.

Notice how programs 1 up to 4 are oscillating in the highest frequency possible and how program 5 reacts slower just because of the order of instructions. When writing control logic, you must be aware that the order of instructions will be able to slow down the maximum speed of decision taking.

In this paper I propose an implementation technique based on clustering of instructions in a specific order to be able to guarantee this maximum speed. When proving the correctness of a control program, my approach will prevent the combinatorial explosion of investigating all possible permutations. When dealing with separate objects there is too much to investigate.

Appendix C: HCADwin tool

HCADwin is written in Delphi (Pascal developing environment) solely by the author. HCADwin is a drawing tool for Functional Control Diagrams. Normally IEC 1131-3 symbols are used, but the symbol library can be configured or modified for each individual project. Every symbol contains configurable Pascal statements. These statements are used when a complete project is translated into a Pascal library unit (to create a DLL[10]). This enables HCADwin to test simulate the logic and control with high performance. The (fixed) cycle time is 100 milliseconds, corresponding with the average cycle time. (PLC: 10 thru 200 milliseconds; DCS: 250 thru 500 milliseconds)

A built in simulation language, HSL[11], can be used to model the physical process. This HSL is using equations to express the laws of nature. When integration is needed some differential equations are added.

The code generation is not limited to Pascal. Structured Text to program a PLC directly, or Java for a standalone application, can be configured.

HCADwin was used to implement a huge real time training simulator for a coal fired boiler connected to a steam generator producing 680 MW electrical with 260 Bar steam pressure. The simulator contains 1100 valves and pumps and 80 PI(D) controllers. The HSL was designed to solve the three phase problem of supercritical water. On a normal laptop the performance of 15x real time is possible. In total 206.000 variables were handled for the control logic and physical models.

Modeling tool like MATLAB or Simulink have poor capabilities for the logic control. This is the normal characteristic for simulating tools. When the logic control can be simulated, like many soft PLC programs (mostly part of the developing tools belonging to a certain PLC brand) the field simulation is absent or poor. HCADwin is unique in combining both.

For state transition diagrams only two symbols are implemented. The “State” symbol holds the unique State Boolean (A number prefixed by the State Variable name) and three time parameters. The “T_Adm” symbol holds the same SV and performs all timing and testing against the actual state time parameters. By “delaying” part of the Pascal instructions associated with the “State” symbol, my implementation rules for clustering the instructions are obeyed. Without this “delaying” the decision speed of a state transition diagram would take multiple cycles.

Without David Knuth [9] the performance of the inbuilt database engines and scripting languages would suffer. His balanced binary tree is frequently used. And I know how to sort without any comparison.

[10] DLL= Dynamic Linked Library. A DLL contains executable code for Windows.

[11] HSL=HCADwin Simulation Language

Appendix D: Example implementation

To illustrate the process control using my state diagrams, a small process is selected. The physical process is a storage vessel with water. The vessel buffers some water to enable the on-off pump to feed another process. Fig. 4 shows a schematic of the physical process:

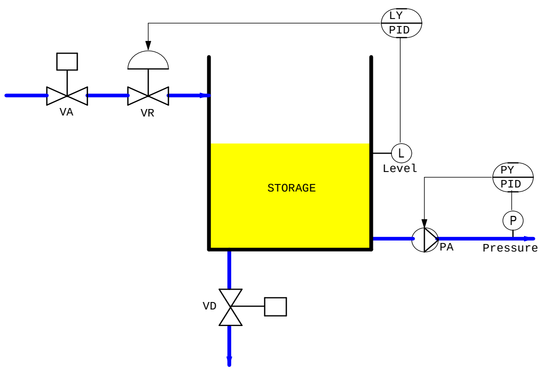


Fig. 4. Hybrid control example. Block valve VA enables the water supply. Control valve VR is used to control the water level at 75%. The level is measured by input L. Block valve VD is used to drain the vessel. The pressure is measured by input P. Water consumption by the processes downstream will drop the pressure. A low/high pressure will start/stop the pump.

Analyze this process. How many parallel processes do you see? I count 2 parallel processes. The main process is the storage vessel. It is a single entity which can be empty, full, draining, controlling or in any other disjunctive state. The valves are used to influence the physical state. The second process is organizing and controlling the pump function.

> When using 5 parallel pumps, it is on a high level just a single pump with 6 operating levels (0, 1, 2, 3, 4, 5 pumps running). Please keep it that simple and add an organizing diagram. Each pump will get its own diagram. When reading an article about a steam boiler and HYTECH[10], I noticed a process representation introducing the number of running pumps as physical states (inducing combinatorial problems by choice and mixing physical and virtual states).

I return to the main process, the vessel. How do we start analyzing it? Assume the system is shut down. No activity. What about the vessel in this state? It could be at any level, but I recognize the “Sleep” state: all block valves are closed and the control valve could be in any position. The second state is the “Drain” state. For maintenance it is desirable to empty the vessel for cleaning. All valves closed, except VD. We enter this state by an operator command: DRAIN_CMD. How long do we need? Maybe infinite, for maintenance cannot be timed with a fixed parameter. I need a DRAIN_STOP command from the operator to return to sleep. The operator initiates the system to work by a VESSEL_START command and we assume that everything will now operate automatically until a VESSEL_STOP command orders us to sleep again. The vessel should hold about 75% level to operate normally. But it could be empty (L<5) or full (L>98). When full, the VA and VR must be closed. When empty, the VA must be opened and after, L>10, the normal operation may continue.

The process can be implemented using state diagrams and obeying the rules for clustering the instructions.

D.1 VESSEL clusters

During and after analyzing the process the state diagrams are created. The resulting implementation is presented here in the proper cluster order.

The transition logic is shown in Figure 5. The VESSEL_A, _B, _R (from Fig. 12) are used for operator commands. The connecting symbol ("Flank") detects a False to True change and is True for only a single cycle. DRAIN_CMD is therefore True on the rise of VESSEL_R. Using single shot commands is a good practice. When a failing Input is used for a command, it can not abusively fire continuous commands.

Read these schematics from left (named inputs) to the right (named outputs). The combinatorial logic is in the middle. Thin lines connect Boolean signals, thick lines transport integer values (indicated with "I"), or real values (indicated with "R", e.g. next to LEVEL).

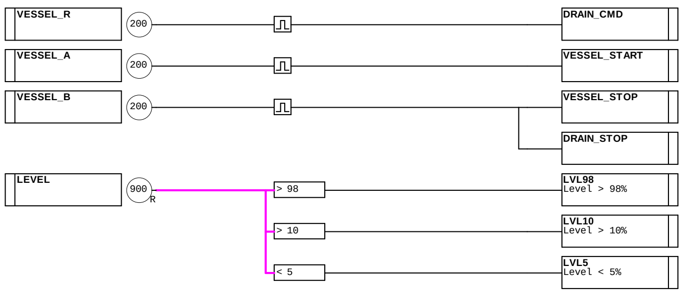


Fig. 5. (100) VESSEL Transitions

The "(100)" above indicates the HCADwin page number. The drawn symbols are translated into a serial list of instructions following some simple rules:

1. Every symbol translates to one or more program lines.
2. The order is: Page by page from low to high page numbers
3. Each symbol has an "execution point" on the upper left corner.
4. Symbols are ordered from the left to the right (and top to bottom if left coordinates are equal)

A fundamental exception is made for "State" symbols (Fig. 6.): These instructions are divided into a first and "delayed" part. The first part is using the state and transition Booleans to write a new state variable (transition cluster). The "delayed" part is inserted at the end of the page (or after the "Flush" symbol just on the right of the last "State" symbol) and creates the state booleans depending on the actual state number (redundant representation cluster). The actual state Boolean is used to copy the parameters to global variables for the timing symbol. Now HCADwin fulfills my rules of clustering the instructions.

Important remark: Implementations with all instructions clustered by symbol boundaries, can never obey my clustering rules and will always need at least two cycles for the decision taking in case of upstream connected symbols.

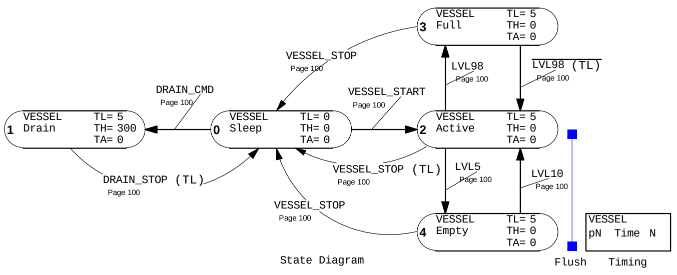


Fig. 6 (200) VESSEL State diagram

The HCADwin representation of a state diagram is shown in Fig. 6. The topology of the diagram is loosely connected with the physical process. (vertical ordering of “Full”, “Active” and “Empty”). Notice the three time parameters for each state. And that “(TL)” is added to the transitions which must wait for the indicated TL=5 seconds (TL from origin state) to be activated. Notice also the inverted transition LVL98 from VESSEL3 to VESSEL2. All transitions show the creation page number for easy referencing. (All transition Booleans to be found on page 100 = Fig. 5.)

And look at the rectangle on the bottom right. All timing for the VESSEL state diagram is performed here. Because all states are disjunctive, a single time-variable (VESSEL_T) is used for the whole diagram. This single timing symbol shares the SV name with all states and is positioned after (to the right) the last state. This “T_Adm” symbol contains the complete timing cluster.

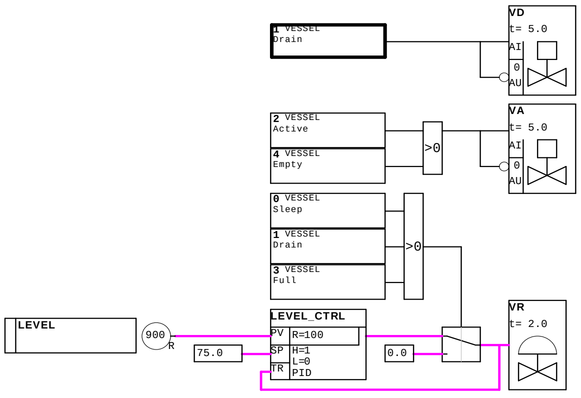


Fig. 7. (300) VESSEL Associated outputs

The VESSEL state diagram is completed with the control logic of Fig. 7. Now an OR (“>0” symbols) of the relevant states is programmed to establish the output pattern. Notice on the right side three symbols with the names of the valves, VD, VA and VR. The attached input signal is effectively the Output of the PLC. Within HCADwin this symbol is used to simulate such a valve. A parameter (t= 5.0) is used to indicate the travel time from closed to open. The global, implied, variable VDPOS holds the position of the VD valve represented by a Real number from 0.0 thru 1.0.

The VD and VA valve are using two inputs, one normal and one inverted. Ignore that now, because it is only useful when a valve can hold its position if both input parameters are false. A normal block valve is opened, closed or traveling.

Notice that reading the VESSEL1 is different (a thicker rectangle is used) from reading the other VESSEL# state Booleans. VD is energized if VESSEL1 And (VESSEL_T<VESSEL_TH) is True. The drain state may take infinite time, but the corresponding output is limited to the first TH=300 seconds. For this process that was not necessary, but it shows the elegance of the method.

The LEVEL_CTRL symbol is a simple PID algorithm, based on the velocity form (Stephanopuolos [11]). Observe how the output of this PID controller is manipulated by a “choose”-symbol. The Boolean on top pushes the switch down when True so the 0.0 is used for the desired valve VR position. To inform the PID controller, the ultimate output is returned on the TR (Tracking) parameter. The next cycle this PID symbol tries to increment or decrement this value. Using this simple method prevents integral windup and is always smooth if needed. Again the VR valve symbol is for simulation.

The three Fig. 5, 6 and 7, are the complete program for the VESSEL state diagram! Each symbol generates some lines of Pascal (see also HCADwin appendix C).

Hybrid control: Note that the state diagram is used to close the VR valve by manipulating the proper output. The PID controller remains active, but is overruled (and informed through TR). When the diagram has no need to force VR, the normal PID controller will take over bumpless without integral windup (due to lacking integrator as a consequence of the velocity form) and starts controlling. The hybrid control comes very natural in conjunction with the state diagram.

D.2 PUMPA clusters

The next part of the program is the diagram for the pump process, see Fig. 8 and 9. I choose input P for the pressure (simplified model for this example) to decide the increase or decrease of the amount of water to be pumped out of the vessel to all consumers. If water is consumed, the pressure will drop. The PID controller, PRESS_CTRL, is “controlling” this pressure and tries to open some virtual valve. When the output signal of this PID controller reaches 75% (0.75) it is a request to start the pump. (<0.25 requests the pump to stop). Now you will understand the logic of Fig. 8 to create the PUMPA_ON and PUMPA_OFF transitions.

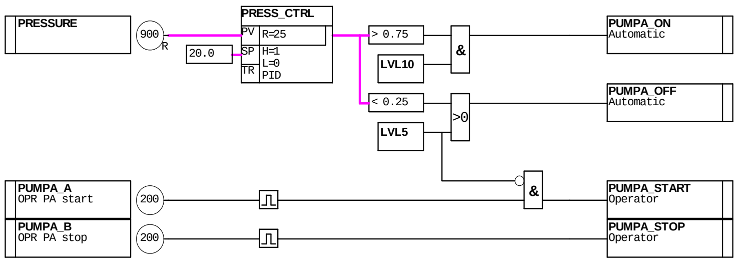


Fig. 8. (101) PUMPA Transition logic

A pump could be damaged if no water is available, so when the level is lower than 5% (LVL5) the PUMPA_OFF is forced. The pump is not working well when it pumps out of an empty vessel, so LVL10 is used in conjunction with the start pump demand. To give the operator an opportunity to interfere with the pump process, he is given some commands also. Note that he is allowed to start the pump above 5% level.

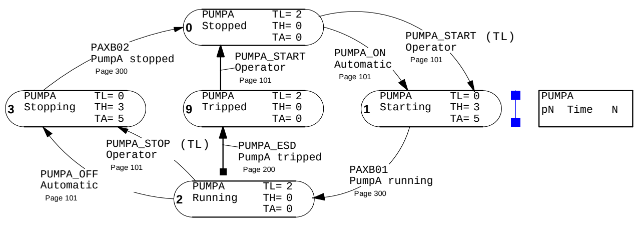


Fig. 9. (200) PUMPA State diagram

Fig. 9 is typically for on/off equipment. The two states, PUMPA0 en PUMPA2, could take infinite time. (TH=0 and TA=0 indicate that no relevant duration can be tested). Because a pump cannot be started without some elapsed time, the PUMP1=Starting state shows the representation of that physical phenomena. The pump is assumed to start within 2 seconds, so an alarm signal could be given after 3 seconds. When it takes 5 seconds, something was really wrong. Note the PUMPA_ESD transition, coming from nowhere (small black square) and forcing the PUMPA9 trip state.

A realistic pump normally has some extra inputs to show malfunction, such as a thermal trip signal when the electro motor is overheated. That would be a splendid candidate to include in the trip transition. The trip time Boolean (PUMPA_ESD) is made in the general (rectangle) timing symbol on the right.

By convention the trip state is left after a deliberate operator command. I choose the PUMPA_START to make the pump available again.

The PAXB01 and PAXB02 transitions are produced by the pump simulation symbol PA in Fig. 10. Normally these “Pump is running” (XB01) and “Pump is stopped” (XB02) signals are returned inputs from the physical pump. The simulation symbol uses a PAPOS Real variable from 0.0 thru 1.0 (comparable with a valve position) to indicate how fast the pump is running while starting and stopping.

The “t = 2.0” parameter indicates that it will take 2 seconds to start the pump. The XB01 and XB02 signals are both false when the pump is traveling.

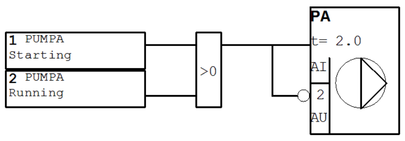


Fig. 10. (301) PUMPA Output

Fig. 10 shows the logic for the output for PA, just a simple OR (“>0”) with the states PUMPA1 and PUMPA2.

Manual/Auto navigation: Make a special note of the parallel transitions from the operator, PUMPA_START and PUMPA_STOP commands (Fig. 9). This enables the operator to manipulate and control as he likes. When stopping a pump, the process will react on it by lowering the pressure, maybe the reason to start another (parallel, but not in this example) pump automatically. The Problem 1 functionality is captured here more elegantly and transparently!

No Manual/Auto needed and always in sync with the diagram.

D.3 Field simulation

It is good practice to test the control logic. I always do that in an incremental fashion. When all clusters of a state diagram are finished and the outputs are sent to the simulated valves, these valve positions can be used to simulate some response of the physical process in order to test the PID controllers or the response on switching values on the analogue inputs.

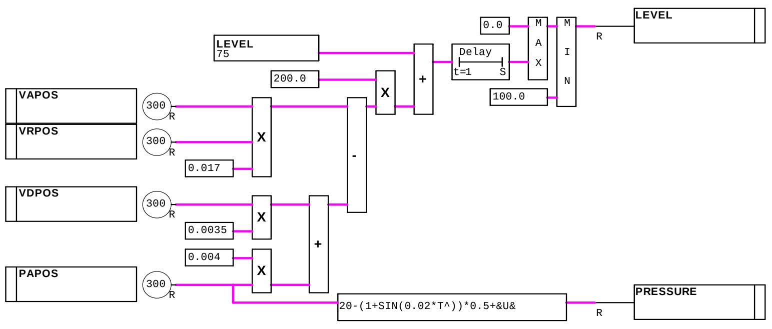


Fig. 11. (900) Simple field simulation for LEVEL and PRESSURE

Using HCADwin to test the control logic, the above field simulation is added. The goal is not to represent the exact behavior, but to make a plausible response. The LEVEL is incremented (upper “+” symbol) by the difference of the mass through VA and VR[12] and the mass loss by draining (VD) or by pumping. The delay symbol is used to prevent algebraic problems when testing PID controllers (and modeling the inertia of the physical process). The MIN and MAX are used to clamp the level between 0 and 100%. The pressure is some fantasy formula using the SIN of the elapsed simulation time (T^) to simulate a periodic consumer behavior. The “&U&” (symbolic input parameter) in this expression equals the connected PAPOS.

D.4 Operator visualization

On the operator screen is a symbolic picture (like Fig. 4) of the process available. When he double clicks in this picture he expects the buttons of Fig. 12:

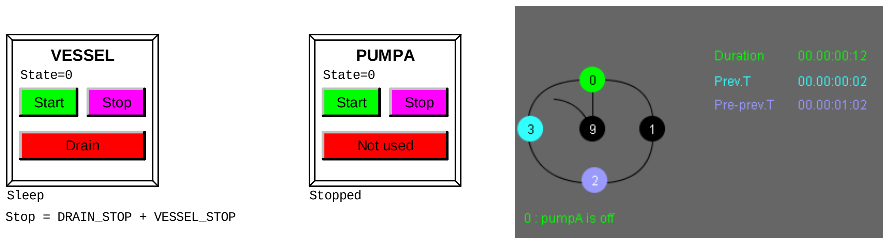


Fig. 12. (200) Operator faceplates and State diagram visualization

The VESSEL_A, VESSEL_B and VESSEL_R signals are connected to the buttons in the left operator faceplate from Fig. 12. These faceplate symbols are normally placed on the same HCADwin page as the corresponding state diagrams.

The state diagram is identical to Fig. 9, but it shows recent visited states as a colored trace. The green “0” state is the current state with an elapsed time of 12 seconds (Duration). Previously state “3” was true for only 2 seconds (Prev.T) and before that state “2” took 62 seconds (Pre-prev.T). Matching colors are used. If the trip state (“9”) would be active, it is shown in red to alert the operator.

This is the important function of the timing symbol next to the state diagrams. The operator can not only see in a glance what the current state is (0: pumpA is off), but also were it came from and how long ago.

In particular when something fails this information is very revealing. To visualize a trace of three states in color is nice when a trip state is activated. The elapsed times and previous states are telling the story in a nutshell.

[12] Multiplying VAPOS and VRPOS does not leave the 0.0 thru 1.0 range. Serial placed valves can be seen as a single valve. Using mass is deliberate, for there is a mass preservation law. Using volumes is asking for non linear behavior when the medium enters another phase.

Appendix E: Timing implementation

An important addition to the state diagrams is made in the timing cluster. Every state number can be associated with three time constants. By adding a single elapsed time variable PROC_T to each named diagram, we can construct an elegant timing program:

```
If PROC1 Then Begin {Also for PROC2, PROC3, etc}
PROC_TL:=PROC1TL; PROC_TH:=PROC1TH; PROC_TA:=PROC1TA
End; {Copy timing parameters}

If PROC_V<>PROC Then Begin {detect transition}
PROC_PP:=PROC_P; PROC_P:=PROC_V; PROC_PPT:=PROC_PT;
PROC_PT:=PROC_T; PROC_T:=0.0; End;

PROC_V:=PROC; PROC_T:=PROC_T+DT; {Save SV and increment
timer with program cycle time DT}

PROC_L:=PROC_T>PROC_TL; {create 3 timing Booleans}
PROC_H:=(PROC_T>PROC_TH) And (PROC_TH>0.0);
PROC_ESD:=(PROC_T>PROC_TA) And (PROC_TA>0.0);
```

The above Pascal statements show how easy it is to implement a general timing for each diagram. Using PROC_L in a transition expression in conjunction with a state having a positive minimum time (PROC1TL) assigned to it, is very convenient. PROC_H becomes true if the elapsed time in the actual state is alarmingly long.

It could be used in the output cluster not to energize a certain Output. Typically done when a vessel is being filled and some “full” indication was expected but not coming. It is wise to close the fill valve. The PROC_ESD is used as a trip condition. A state took much too long and an emergency measure has to be performed, like entering a trip state.

Note that by assigning a zero to the time parameter, its function is omitted. The copies of previous state numbers and elapsed times are used for visualization only.

When these timing Booleans are used with some skill, the state diagram starts to become “self repairing”. If some Input fails (or the process acts funny), this abnormal behavior is captured by some other Input AND PROC_H (using a normal or a deviant route through the diagram).