CONTROL SYSTEM EVOLUTION AND THE IMPORTANCE OF TRIAL AND ERROR

P. Duval, M. Lomperski, DESY, Hamburg, Germany
J. Bobnar, Cosylab, Ljubljana, Slovenia

Abstract

In this paper we address the importance and benefits of trial and error in control system evolution. Here we refer to the control systems of particle accelerators and large machines, whose control systems, although complex, will not lead to catastrophe in case of failure. We likewise focus on the evolution of control system software, although the issues under discussion will apply to and are often driven by control system hardware. We shall contrast classical Darwinian evolution via natural selection with control system evolution, which proceeds rather via artificial selection, although there are numerous software memes which tend to replicate according to their 'fitness'. The importance of general trial and error, i.e. making mistakes and learning from them, in advancing the capabilities of a control system will be explored, particularly as concerns decision making and overcoming *Einstellung*.

INTRODUCTION

A mature accelerator control system will be able to address a wide variety of problems which might arise throughout the controlled facility's natural lifecycle. Solving new problems or a push to provide better solutions to old problems will generally lead to control system evolution, even if this amounts to little more than keeping up with industrial or commercial components. The way one goes about problem solving will in turn have a marked influence on the pace of this evolution. We will discuss many of these aspects below, finishing with a few concrete examples of control system evolution at play.

GOALS AND PROBLEM SOLVING

The God Complex and Einstellung

When we are well-versed in our control system and at the same time faced with a new problem or challenge we are apt to fall prey to the God Complex, i.e. that "no matter how complicated the problem, you believe that your solution is correct." [1, 2] This is furthermore often compounded by what psychologists refer to as the *Einstellung* effect, or the "predisposition to solve a given problem in a specific manner even though better or more appropriate methods of solving the problem exist". [3,4]

The danger is not that our problem won't get solved. It most likely will. The danger is that we might not only miss an opportunity to explore new ideas, we might also end up wasting resources, and/or missing the big picture entirely due to our rush to implement a known solution.

Priming and Anchoring

Indeed our choices in problem solving and decision making are often due to an implicit memory effect known as priming [5], where exposure or familiarity with one stimulus (or solution paradigm) can influence our response to another. The classic trivial example: "How many animals did Moses take on the ark?" (answer: 0) might appear to have little to do with our decision making until we realize that our *expectations* can be easily primed, a case of priming known as *anchoring* [5]. For example, imposing an artificial deadline of one week to try some solution automatically suggests a level of difficulty. Worse, refusing to consider a new solution because "everyone else does it differently" suggests a knowledgeable rejection of the new solution. Unfounded expressions such as "one week" or "everyone else" often serve only to anchor our expectations at some level.

Accumulated Advantage

The previous example of anchoring ("everyone else does it differently") is also an example of the effect of accumulated advantage, often referred to as the *Matthew Effect*, (from Matthew 25:29 in the King James version of the Bible) [6]. In point of fact, our opinions are strongly related to and often dependent on those of others. The crowded restaurant *must* serve better food than the empty one next door! In an experiment by Duncan Watts [6], two sets of college students could download garage band music from two web sites. The sites were identical except that in one case the students could see the *likes* and downloads of everyone else. It's not surprising that in one case there were a handful of hit songs and in the other the *likes* and downloads showed a flat distribution.

Trial and Error

We should in any case be aware of the aforementioned challenges to our problem solving abilities. Whether we admit that we already know the solution to a new problem or not, the practice of trial-and-error cannot be avoided. The basic algorithm of trial-and-error can be described as:

1) Define what constitutes a solution to our problem.
2) Try *something*.
3) Check to see if the problem is solved. If not:
4) Modify *something* into a more promising direction and repeat step 3). Or, if the problem is solved:
5) Quit.

ISBN 978-3-95450-189-2

The psychological effects we have just discussed will of course influence the *something* that we initially try in step 2). In fact, if there is any kind of time pressure the best bet is indeed to go with our best over-all hunch. If on the other hand there is a time-window for bold experimentation, trying several *somethings* might lead to remarkable improvements.

EVOLUTION

Evolution, Darwinian or otherwise, naturally progresses via trial-and-error. The incremental evolutionary steps might occur by chance, as in the case of the natural world, or by design as in the case of control system evolution. Either way, the measure of success is the ability to replicate. Thus there is an implicit drive to improve.

Darwinian Evolution

In Darwinian evolution [7] the replicating unit is the gene. Any mutation which leads to a greater chance of survival will in turn lead to an organism's genetic material replicating itself more often, which is what we mean by improvement. Evolution by natural selection is of course slow and has a direction. That this occurs by chance means that evolutionary changes cannot be reengineered in order to improve performance. Any improvement comes entirely by trial-and-error.

An oft cited *proof* of *intelligent design* by creationists is the eye, which is so complex that it couldn't possibly have happened by chance and *must* have had a designer. A billion years, though, is ample time for cells initially able to only distinguish dark from light to successively evolve into such a remarkable organ. More to the point, the design is actually rather clumsy, as recognized already in the 19th century by Hermann von Helmholtz. No engineer would route the wiring leading from a camera's photo cells back into the path of the light source and then bundle it all into a thick cable near the center of the collecting surface, thereby creating a blind spot!

There is no chance to reengineer this into a more sensible solution. Nor is there any chance that a random mutation will fix the design flaws.

It is nonetheless instructive to recognize how incredibly well the eye does work (in tandem with our visual cortex). The design flaws are practically irrelevant, a point which should be remembered when the temptation to refactor complicated, yet well-working, software arises.

Software Evolution

Software evolution is driven by design decisions from the very beginning. Although survival might still be of the fittest, the agent of change is artificial- rather than natural-selection, and the replicating unit will be the *meme*, the smallest idea that gets transferred within a culture [8] (e.g. the idea of *sockets* or *threads*, but not necessarily the implementation of them).

Manny Lehman identified three categories of software, S- (*specific*) programs, P- (*procedural*) programs, and E- (*evolutionary*) programs [9]. S-programs are written once for a specific purpose. P-programs implement a set of procedures only (e.g. play chess). E-programs perform some real-world activity and adapt to the environment and circumstances in which they run.

As much as we might wish particle accelerator control systems to be P-programs, they are in fact E-programs and necessarily evolve. In fact, as the environment in which a control system operates does indeed evolve, one of Lehman's Laws [9] asserts that the quality of the control system will decline unless it also evolves.

Here, however, we do have and often utilize the ability to refactor *bad* or *clumsy* design decisions. Of course, the question remains as to what a bad design is and (like the eye) as to whether it is in the end worth the risk of changing a (well-running and complex) running system merely for the sake of improving the design.

Complex software might also contain vestige routines (analogous to the appendix) which have no practical purpose but continue to be accessed by vintage application programs and are therefore required to exist. Thus, API breaks in reusable software such as control system libraries should be avoided when possible, including the disposal of deprecated API routines or class methods, unless the consequences of doing so are understood beforehand.

In addition to keeping pace with an evolving hardware environment, control system software will evolve on its own accord in order to improve or introduce functionality. The pace of evolution here will be strongly dependent on the developer's susceptibility to the psychological effects mentioned in the previous section.

Regardless of pace, any real evolutionary change will occur via trial-and-error. Typically, coding modifications will be run through various unit tests (a tight trial-and-error loop) until there is a new release candidate. The next trial might occur in the field when the software is deployed. After deployment, however, the cost of error will be much higher. In the case of accelerator control there is fortunately little or no chance of catastrophic error (as there is in airplanes or nuclear power). Nonetheless an error can lead to downtime or damage to equipment. Thus the cost of error should be examined along with rollback strategies prior to any new deployment.

EXAMPLES

Control System Protocol

One of the problems we sometimes have to deal with is an unacceptably high load (CPU and/or network) on a control system server. If this load is primarily due to information transfer from server to client then we have an issue with the control system protocol.

The TINE [10] control system makes use of the device server paradigm, where a server exposes control system elements as instances of *devices* and offers access to their attributes and actions through *properties*. It also offers *publish-subscribe* data acquisition, which in itself goes a long way in reducing unnecessary load on a server due to data transfer to multiple clients. However if client applications obtain data via repetitive non-persistent

ISBN 978-3-95450-189-2

transactions (*polling*) then there is no load reduction from *publish-subscribe*. The situation can be compounded many-fold if a server with many device instances (e.g. a vacuum pump, beam position monitor, or power supply controller - PSC) is requested to deliver information from all elements one-at-a-time. And precisely this is an all too common occurrence with simple client panel applications.

When faced with this situation in 2009 with multiple *ddd* [11] clients accessing the FLASH PSCs we decided to eschew the traditional split-the-load-among-multiple-servers approach and add a new feature to the control system protocol called contract coercion [12].

As the PSC server is not only prepared to, but prefers to send property information for all PSC instances for a given property as a multi-channel array, we addressed the question "Can we coerce a client's synchronous request for the value of a property into a monitor for all values of that property?" We answered yes and then introduced contract coercion. The initial results were more than encouraging as the load on the server was effectively decimated without modifying a single line of client code.

To be sure, there was a significant amount of tight-loop trial-and-error with unit tests, etc. prior to deployment, but we were nevertheless aware of the costs of unforeseen errors beforehand. We assumed, based on prior testing, that the likelihood of a serious error on *most* clients was extremely small. If an exotic client did have an error we based the decision to rollback or not on how critical to operations the exotic client was and whether the error appeared immediately or some significant time later.

In the end, no rollbacks were ever necessary. Several errors were nonetheless encountered and repaired in the months following initial deployment. Likewise, the ensuing years saw several quality-of-service additions to the initial contract coercion implementation, each with its own trial-and-error process.

Control System Services

The TINE control system offers many central services, among them a plug-and-play system concerning name resolution.

The TINE Equipment Name Server (ENS) maintains a device server database and provides address information when a client needs to contact a control system element. The ENS database can of course be modified by an administrator, but in general it is updated automatically. The plug-and-play mechanism will add a new server to the database or update a server's meta-information with every server start.

This level of automation requires a good deal of trial-and-error whenever new features are added. The ENS must not only guarantee a unique entry for a control system server it must also inform any server trying to usurp an existing name that its request was denied. The requested names and meta- information of any new server must also be validated, etc.

When we make modifications here, however we are modifying a central service rather than the control system protocol itself. What are the costs of error in this case?

The ENS would appear to offer a critical service, in contrast to, say, central archive or alarm servers. In fact it is *semi*-critical. All clients already have a fallback mechanism (using the last locally cached address in the event of address resolution failure) when the ENS is not operational. The worst that can happen is either 1) a new server will not be able to plug itself into the system, or 2) an existing server starting on a new host will not be able to modify its address information.

The other TINE central services are even less critical in that operations are never threatened in the event of error.

Applications

A good example of a specific application with on-going trial-and-error is the Operation History Viewer in TINE Studio [13, 14]. This application shows the machine state information (including problems) over any selected time range. The *problems* state indicates non-availability of the machine and can be divided into sub-systems, where one has the ability to browse through the fatal alarms responsible for the downtime. The goal is to have a fully automatic calculation of operation and availability history. As blame for non-availability is assigned to fatal alarms, we see that the application consists of more than a mere presentation of data, and involves, among other services, the central alarm system in a vital way. We may not break free from the trial-and-error loop here for some time to come and have added the ability to post-correct both the state information and the availability information (by a machine coordinator) over any time interval.

The costs of error here, as for most applications, apply almost exclusively to the application itself, and will have no impact on operations unless the application is critical to operations (which this isn't). This is not to say that we can make errors with impunity. Once an application is regularly used, any degradation in quality of service will of course result in unhappy customers. Ensuring that the most-used features continue to work properly is generally sufficient to allow deployment of a new version. Should an error be discovered, a rollback can easily be made while the error is dealt with.

CONCLUSIONS

Control system evolution will occur if for no other reason than the necessity of keeping pace with the commercial and industrial world. Real innovation in control system software will involve a trial-and-error period. This period can be extensive or even continual, but primarily constitutes what is meant by control system evolution. Dramatic improvement most often occurs if we resist the psychological pressures to solve any new problems in a tried-and-true manner and admit that we perhaps don't already know the best course of action. Often enough, deadlines will require us to play it safe, but if we have a large enough time window for development such that we can test several solutions to the same problem then we can often make great strides in the advancement of our control systems.

ISBN 978-3-95450-189-2

REFERENCES

[1] A. Cochrane, https://en.wikipedia.org/wiki/Archie_Cochrane

[2] T. Harford, “Trial, Error and the God Complex”; https://www.ted.com/.

[3] A. Luchins, "Mechanization in problem solving: The effect of Einstellung". Psychological Monographs, 1942.

[4] M. Bilali and P. McLeod, “Why Your First Idea Can Blind You to a Better One”, Scientific American, March 2014.

[5] D. Kahneman, “Thinking Fast and Slow”, Penguin Books, 2011.

[6] S. Pinker, “The Better Angels of Our Nature”, Viking Press, 2011.

[7] R. Dawkins, “The Blind Watchmaker” (and references therein), W.W. Norton & Co., 1986.

[8] R. Dawkins, “The Selfish Gene”, Oxford Press, 1976.

[9] M. Lehman, https://en.wikipedia.org/wiki/Software_evolution

[10] TINE, http://tine.desy.de

[11] jddd, http://jddd.desy.de

[12] P. Duval and S. Herb, “The TINE Control System Protocol: How to Achieve High Scalability and Performance”, in *Proc. PCaPAC’10*, paper WECOAA02.

[13] P. Duval, M. Lomperski, and J. Bobnar, “TINE Studio, Making Life Easy for Administrators, Operators and Developers”, in *Proc. ICALEPCS’15*, paper WEPGF133.

[14] P. Duval, M. Lomperski, H. Ehrlichmann, and J. Bobnar, “Automated Availability Statistics”, presented at PCaPAC’16, Campinas Brazil, Oct. 2016, paper WEPOPRPO18, this conference.

ISBN 978-3-95450-189-2