RECENT TRENDS IN ACCELERATOR CONTROL SYSTEMS

I. Verstovšek[#], F. Amand, M. Pleško, K. Žagar, Cosylab, Ljubljana, Slovenia

Abstract

Control Systems play a central role in the daily operation of accelerator facilities. They also pose unique challenges during the overall design and realization of the machine. This paper reviews important recent trends in the field of accelerator control systems and describes how they address the increases in complexity and scale of their host systems. Selected topics are the various software frameworks & hardware platforms and the use of FPGA's. Also considered are design methodologies and management processes. We'll illustrate the role the control system plays as a unifying force, not only for sub-parts and technologies, but also on project teams: it supports the shaping, very early in the project, of subsystem responsibilities and system requirements.

INTRODUCTION TO THE CONTROL SYSTEM AND ITS COMPONENTS

Control Systems (CS) of the so called big physics facilities, some of men's most complex and costly machines, are not only highly complex systems in their own right, they also come in a great variety, differing in many fundamental aspects. Still, one can come up with a reasonable definition of a control system:

A control system is a set of interconnected computing hardware and I/O peripherals, with dedicated programming, that allows operators and scientists to bring the entire machine in a desired operating state in a safe, controlled and predictable manner, such that the experiment can be performed and observational data collected.

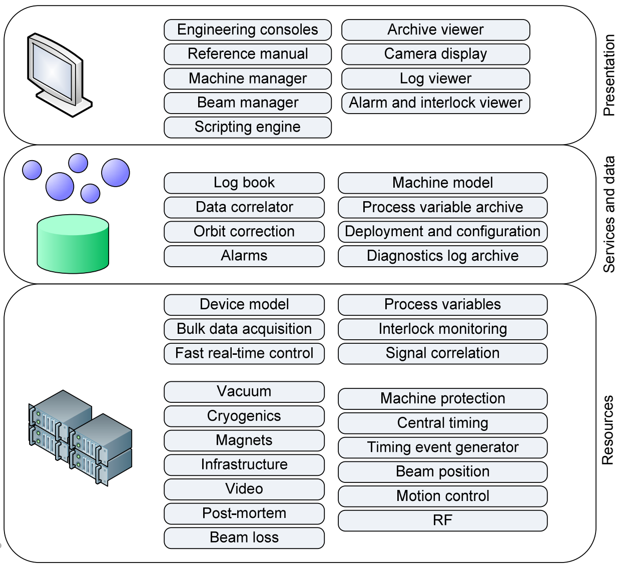

Figure 1: Control system overview.

As such the CS can be seen as (delivering) a service to the experiment and experimenter. We want to stress that it is important to consider the CS through this user oriented paradigm when designing the overall system, rather than merely looking at the physical - and software components. System architects refer to this paradigm as the *use case* - or also *conceptual view*. This as opposed to e.g. the physical or deployment views [1, 2].

From this use case or user point of view one can perceive e.g. the following needs:

- Persistent storage and retrieval of data such as configuration parameters, run-time loggings,…
- Accelerator Physics applications that can execute e.g. setup sequences.
- Interactive user interfaces for engineering and operational purposes.

To successfully implement solutions for these needs one has to carefully consider the following questions:

- Which of these needs pose hard real-time requirements on (parts of) the machine and which don't?
- How to define subsystems such that their responsibilities are clear and have little overlap (high cohesion) and that the amount of their interconnections can be kept to a reasonable minimum (low coupling)?

To illustrate the last point we look in more detail into two important subsystems, the Timing System (TS) and the Machine Protection System and their inter-relations. The TS orchestrates all actions in the machine through the distribution of high resolution time stamps. The MPS detects problems and triggers a fast machine shut-down to minimize damage to the equipment. The two are tightly coupled, see e.g. [3]: *"The MPS uses the timing system to assure operation within specified duty cycle and pulse width limits. In addition, information on shut-down causes originating with the MPS is broadcast by the timing system for post mortem analysis."* Sometimes, a "beam permit" subsystem is defined as part of the CS. One can easily image different interpretations as to which subsystem does exactly what in this case. Such misalignments have to be sorted out early in the project to prevent them from becoming liabilities later. This is a people management issue, more than a technical one. The centralizing role of the CS in the entire system places the CS team in a natural role for clarifying and documenting such subsystem responsibilities. We will explore this aspect in more detail in the development processes paragraph of the paper.

[#]igor.verstovsek@cosylab.com

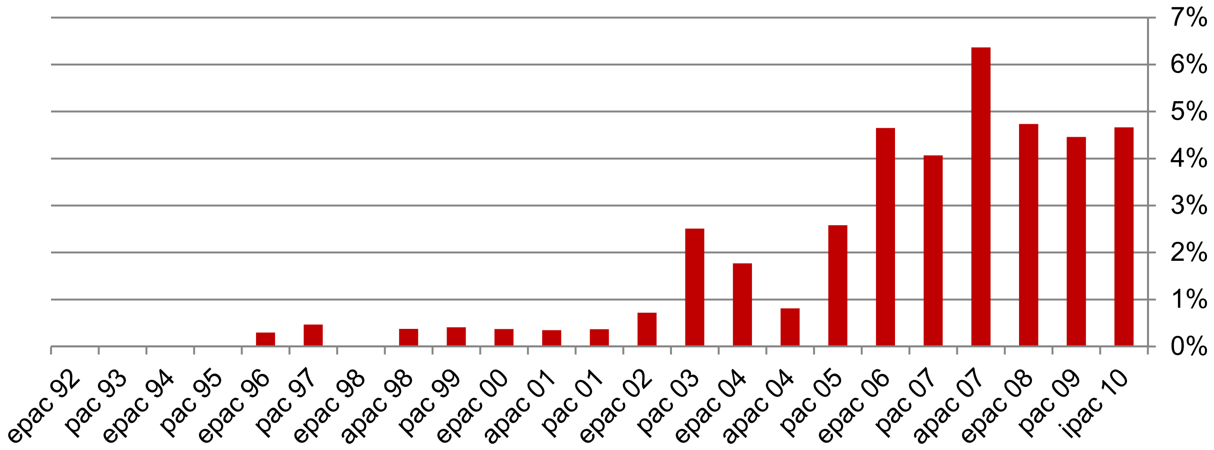

Figure 2: String "FPGA" in conference articles.

THE USE OF FPGA'S

When we speak of FPGA's we essentially mean reprogrammable hardware. The logical behavior is not defined during mass production of the silicon nor board assembly in the factory, but rather in-the-field, e.g. as part of system integration. FPGA's possess many of the typical advantages of implementation in hardware versus in software: faster, deterministic (synchronized) and with (true) parallel execution of logical operations. At the same time they exhibit typical benefits of software realizations: high-level and flexible definition of behavior and the ability to redefine it, either for requirements changes, stepwise refinements or simply to fix "bugs". Applications in CS design are digital LLRF control (for RF cavities), the MPS and timing systems, Data Acquisition and many more.

Figure 1 illustrates the increase in interest and consideration of FPGA's in the accelerator community. We simply measured the occurrence of the text string "FPGA" in pac/epac/apac/ipac conferences over the years. One can observe an order of magnitude increase over the last decade.

We do not need to look far for the reasons for success of FPGA technology. FPGA's do exhibit best-of-both worlds properties without a big price tag, at least not in initial purchase price. There are however a number of well-known pitfalls in the software development practice that enter the hardware discipline through FPGA technology. They can turn costly if unrecognized.

A first one is the false sense of flexibility of (re-) programming in the scope of very large systems. No engineer would consider changing structural aspects of a bridge design halfway through the building process. Even in the early stages, people can envision that postponing choices like the suspension mechanism until later makes no sense. In software projects however the impact on cost and risk of late changes can be equally problematic and yet they are not always perceived as such. A level of maturity and experience with large software projects is needed to assess what can be safely altered later and what cannot without bringing the project in jeopardy.

A second one is added complexity. To make the point in a philosophical way: with modern FPGA development tools it is easy (and cheap) to neglect Occam's Razor principle, i.e. in case of equal solutions to a problem, taking the simpler, more elegant one. An exaggeration to illustrate the point: With a few mouse moves one can drag-and-drop a complete CPU onto the FPGA, run a limited version of Linux on the board and demonstrate the proof of concept with some kind of "Hello World!" app… then to discover compatibility headaches you've introduced when making the "Real World!" application.

SOFTWARE BUS OR MIDDLEWARE

The previous paragraphs stated and illustrated managing complexity as a main challenge for CS projects. Abstraction and generalization are classic concepts for achieving this aim and are widely used in software engineering. In distributed systems (HW and SW on a large number of networked computers) an important abstraction tool is the so called *software bus*. Other designations are software *framework* or simply *middleware*. Examples in the CS world are EPICS, CMW/FESA, TANGO, TINE, DOOCS, MADOCA, e.a. The goal of these frameworks is to allow distributed software components to be "plugged-in" into a system analogously to a hardware bus, such as VME. This allows large parts of the software system to treat, in our case *control*, these components in a standardized way, independent of the different physical equipment they represent.

One might ask whether the various solutions that are available differ in fundamental ways. In terms of technology and certain performance criteria they certainly do. In terms of service they offer to the system they essentially don't. In selecting the right match for your system it is good to take a number of non-functional aspects into consideration, besides the purely technical ones:

- Support: Is there a strong user and developer community for this system?
- Is there a large number of supported devices, with a relatively small set of interfaces? (remember, the

goal of the added layer in software is achieving generalization)

- Is it lightweight on dependencies? It should not depend on other complex middleware and should have few central services as single point-of-failure.

EPICS & TANGO, widely used in the community, meet these criteria. But they are not the only ones to do so, which supports the observation that also other frameworks are chosen to provide the software bus service to accelerator control systems.

HARDWARE PLATFORMS

Also in hardware platforms we observe the clear absence of the winner-takes-all phenomenon that has so often shaped the IT industry (MS Windows, Google search, PCIe). The very diverse and unique needs of experimental facilities are probably accountable for market dynamics that set it apart from the mass consumer IT space.

Still, and we'll get back to this in the next section, one should standardize as much as possible in CS projects. A choice of one of the current HW platform standards (VME, ATCA, cPCI,…) over the others is in order. What should be the criteria for choice? A comparison of 3 major platforms is summarized in Table 1.

Table 1: Hardware Platform Comparison

	VME	ATCA	cPCI
Vendor support	High/ Declining	Low/ Growing	Medium/ Stable
Maturity	High	Medium	High
Longevity	Medium	High	High
Max. transfer rate	VME: 40MB/s VME64: 80MB/s VME64x: 160MB/s VME320: 320MB/s	1Gbps, 10Gbps (Gigabit Ethernet); 250MB/s/lane (PCIe)	PCI: 133MB/s PCIe: 250MB/s/lane (up to 16 lanes)
Topology	Master-slaves	Star Dual star Full mesh	Master-slaves
Form factor	6U (64 bit) 3U (32 bit)	12U (ATCA) 2U (μTCA)	3U
High availability	Medium	High	Medium
Software support (Linux, EPICS)	High	Medium	Medium
Cost	High	High	Medium
Users	SNS, SLS, Diamond Light Source, NSLS II, …	XFEL (LLRF), ITER, TPS (considering)	ALBA, TPS, CERN (LHC collimation), LANL, ORNL, ITER (planned)

Two criteria we believe should receive proper attention:

1. Usability, i..e. what the platform can do, the features, but also how well the relevant tools are debugged.
2. Longevity

Strong performance figures measured by today's standards means the platform will not be outdated in a few years' time, but this is not enough. One should look for a platform that will be accepted most likely by the majority in the industry. That means that one shouldn't look just at other labs. The reason is that a well-accepted technology determines a complete and broad market that not only provides many manufacturers (and thus lowest prices) but also a vast number of users that will be "in the same boat" with you, if you have chosen the same technology as them.

DEVELOPMENT METHODOLOGIES AND PROCESSES

Experimental setups and their control systems are growing in complexity. As experience with accelerators grows and the computing equipment to build them becomes more performing and at the same time economical, the expectations on the machines, for example in terms of flexibility, are also growing.

The expectations can be met, provided we can manage the growing system complexity that comes with it. One natural evolution has been that these projects moved from being a research project in itself to becoming engineering projects. Examples of this are the already mentioned software and hardware standardization. But apart from this it also entails standardizing on managerial aspects of the project. We believe they are more crucial to success then the technological standardizations.

Let's look at the development process. There are a number of different development methodologies that have proven their value (waterfall, SA/SD, iterative, agile …). Yet, they have different vocabularies with on top varying interpretations between development organizations. The trend towards large international projects with in-kind contributions requires the establishment of a common language to communicate about e.g. requirements, deliverables, unit & integration tests e.a., and their expected levels of quality and completeness. In view of the overwhelming amount of such (intangible) artefacts in these projects, this can only be achieved by agreeing on very rigid standards.

Requirements management is a special point of attention. People generally feel a resistance to writing down requirements they know are not final. Still it is better to write them down early, in the form they are (incomplete, inconsistent). This way the design effort will be more focussed and effective, as the pain-points that *must* come up and be dealt with to bring the requirements and specification in their final shape will come to attention earlier! Investing in a tool that facilitates keeping overview of the many hundreds of requirements, their consistency and their traceability to components and test cases is worth considering.

As mentioned in the introduction, the centralizing role of the CS in the entire system places the CS team in a natural role for clarifying and documenting the system requirements. One could introduce the term *meta-control-system*: the CS and CS project team as a tool/leverage in the hands of the project leader to orchestrate and harmonize the overall project effort.

Two more development process aspects we'd like to shortly mention. Realize vertical prototypes from the beginning. These are usage-scenario driven, thin-slices through the (multi-layered) system architecture. They bring the system to live and challenge assumptions that were made in the definition of the interfaces (API's). It also make the system more tangible which helps requirements elicitation (*seeing* a concrete, tangible system has shown to bring up more useful customer feedback then presenting drawing board abstractions).

An example of this is the MACS Vertical Column that was completed in the early stages of the MedAustron project. From a functional GUI, down to actual control signals for a power converter, the single rack operates with the same software layering as the final system.

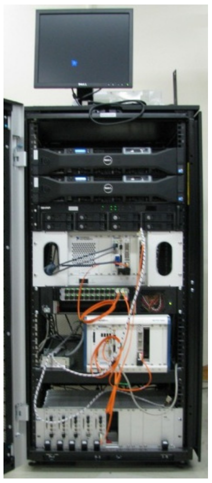

Figure 3: MACS Vertical Column in the MedAustron project.

A related concept is iterative development: evolving from a partial, but completed, working subset to a complete machine, as opposed to the "classic waterfall", with a specification-only start, coding phase in the middle and a "big-bang" integration at the end. Iterative development brings out on valuable customer feedback early and hence reduces risks late in the project.

CONCLUSIONS

The design of control systems for accelerators has seen a shift from being a research topic to becoming a mature engineering discipline. Main concerns have shifted from meeting performance requirements to managing the growing flexibility expectations and integration complexities.

The result is a trend of convergence without over-consolidation:

- There is a set of proven, stable solutions, e.g. in middleware.
- Teams repeat the proven recipes and integrate a growing number of off-the-shelf components, furthering standardization.
- Yet, there remains and will remain enough flexibility to meet exceptional requirements as solution providers understand the specific nature of this market.

It is also in the nature of this field to keep pushing the boundaries of what is considered possible. For CS integration this translates in a continuous challenge to manage the CS integration complexities and deliver low-risk, in-time and within-budget.

REFERENCES

[1] Philippe Kruchten,"Architectural Blueprints — The '4+1' View Model of Software Architecture", Nov. 1995, IEEE Software 12 (6), pp. 42-50.

[2] Matthew West and Julian Fowler, "Developing High Quality Data Models", 1999, The European Process Industries STEP Technical Liaison Executive (EPISTLE).

[3] C. Sibley, A. Jones, D. Thompson, "Implementation and Integration of the SNS Timing and Machine Protection Systems", ICALEPCS´03, Gyeongju, Korea, 13 - 17 Oct. 2003, p.145.