

ACCELERATOR COMMISSIONING TOOLS REVIEW, DESIGN PATTERNS, AND ARCHITECTURAL APPROACHES

W. Sulaiman Khail*, P. Schnizer
Helmholtz-Zentrum Berlin, BESSY, Berlin, Germany

Abstract

Particle accelerators are costly, complex machines. Modern light sources in particular cannot reach design performance without a dedicated software stack for commissioning and tuning, validated during design and construction using Monte-Carlo studies and digital twins. As both the software and the twins grow in scope, architecture matters.

We review representative tool stacks, extract recurring design patterns, and show how treating the system as interactions over a large shared state leads to simpler, more consistent implementations of commissioning tools and digital-twin workflows.

INTRODUCTION

Complex experiments or accelerators require a sophisticated design, build, assembly, and commissioning procedure. We use accelerators as our main example, as we are working within an accelerator environment.

Designing and Building Complex Projects

Accelerators are typically first designed using computer simulations. During the design process, technology experts join the design teams to ensure that the required devices comply with proven technology. If not, appropriate research and development programs are initiated.

Devices are designed and built to provide the intended functionality of lattice elements. This process improves the prediction of parameter variations that these devices will have. Along with this process, the lattice design is revisited and tested to ensure that the accelerator complex will achieve the desired performance parameters. Typically, this check is performed during the device production process as soon as measurements of manufactured devices become available. Finally, after the entire complex has been assembled, it is commissioned: first, its devices, then functional units, and finally with beam. To summarize:

- An accelerator lattice is specified as a set of lattice elements.
- Devices are designed, built, and tested to realize the elements' intended functions within tolerances.
- Devices are assembled, installed, and aligned into the machine.
- The accelerator is commissioned and tuned to achieve design performance.

Commissioning Tools Becoming Part of Operations Infrastructure

Since MAX IV – the first 4th-generation synchrotron light source – has been successfully put into operation [1], subsequent machines that are pushing the limits of this design [2,3] depend heavily on commissioning carried out by appropriate computer-based tools, i.e., self-adaptive systems. The devices of these machines cannot be built and operated with sufficient accuracy. Typically, these machines cannot simply be turned on using the parameter set derived from a dedicated commissioning period; instead, all devices must be carefully tuned to operation. As a consequence, these commissioning steps must be executed periodically – often every time the machine is started up, which is typically every week. Therefore, these tools must also be executed periodically, and machine operation relies on their functionality.

The development of commissioning tools begins concurrently with the design of the machine. The design process not only verifies the performance of the ideal lattice but also ensures that it is robust enough to work with variations in device performance. This means that the design must be either robust enough to work out of the box, or that appropriate commissioning tools must be available to measure the actual machine parameters with sufficient accuracy, so that mitigation actions can be applied to achieve the desired performance.

As modern machines depend heavily on these commissioning tools, they have become critical components that are designed and tested alongside the machine using Monte-Carlo simulations.

REVIEWED TOOL STACKS

We examined several commissioning toolboxes and identified recurring patterns that these toolboxes employ. In this paper we present these patterns and demonstrate how they can be used to structure different aspects of commissioning tools and digital twins.

Our ideas build on our previously published and ongoing work [4–7] together with tools and tool stacks used to design or commission accelerators:

- Johan Bengtsson's tracking code Tracy II and associated tool set [8,9].
- Online models

NSLS II This online model allows simulation of the whole accelerator complex [10]. It uses Tracy II as its simulation engine.

* waheedullah.sulaiman_khail@helmholtz-berlin.de

FLAME for FRIB This online model is based on beam envelope calculations [11].

- Commissioning toolsets
 - The Matlab Middle Layer (MML) code base [12]. This package has been used by many synchrotron light sources and adapted to their needs. It uses the Accelerator Toolbox (AT) as its internal simulation engine, which itself builds on integrators developed for Tracy II [13].
 - The Catap tool stack developed for Clara [14].
 - Pamila, developed at NSLS II [15].
- Simulated commissioning environments [16].
- Measurement orchestration software: Bluesky [17].
- Workflow engines: Reana [18].

We have also been developing our own tool sets, which provide parts of the intended functionality: Berlin Accelerator Control Toolkit (BACT) for providing commissioning tools and dt4acc as a digital twin framework [7].

ARCHITECTURAL CHALLENGES

A review of these software packages revealed that most of them need to address the following challenges:

- Provide a flexible architecture that is easy to understand, can be adapted to specific machines or facility environments, and allows modification or extension of functionality.
- Handle hundreds to thousands of process variables and set-points in a consistent and straightforward manner.
- Provide a simple, unified interface to the different aspects of the tool set.
- Acknowledge that sophisticated software packages must be interfaced with, and that these packages often use input formats that differ significantly from those of real-world devices.
- Describe complex measurement, tuning, or commissioning procedures independently of the specific devices used, as far as reasonably possible.
- Provide a consistent interface for the users of the tools. Some users require graphical interfaces, while others rely on well-designed scripting interfaces.
- Support handling multiple similar but distinct versions of the same machine, for example to enable swapping components with spares, accommodating machine upgrades, or supporting Monte-Carlo studies.

In the following sections, we present patterns that address these challenges and cover different aspects of commissioning tools and digital twin environments.

PROVIDING A CONSISTENT INTERFACE: THE UPDATE PATTERN

Building commissioning tools or digital twins requires handling a large set of process variables. These variables are typically named using short, mnemonic conventions. However, it is often realized that only some values are changed from the total state set. Furthermore, usually only the value of a specific property of a device is changed, leading to the update pattern as follows:

```
update(dev_id, prop_id, value, on_failure)
```

The device ID and property ID should be chosen to be straightforward and easy to understand for those who use, develop, or maintain the codebase. If many values need to be updated consistently, the different updates can be grouped together into a transactional update (see [4]).

It is a deliberate design choice to expose only what is necessary to the outer layers of the system. Many dedicated commands and parameters are typically required only to configure a simulation tool at startup; thus, only a limited set needs to be available for the tools discussed in this paper. The update pattern allows a consistent approach: for example, in a start-to-end simulation that employs different simulation codes for different subsections of the accelerator, a unified interface can be provided based on the update pattern. Implementation details are then hidden behind the system facade. Similarly, swapping different codes would only affect the system facade, without impacting any other parts of the code.

SIMULATION CODES INPUT AND DEVICE PARAMETERS

Accelerator simulation codes are typically based on the paraxial approximation. They compute only deviations from the ideal orbit. Furthermore, almost all inputs describing the fields to which the particles are exposed are independent of particle energy. Device inputs, however, are given in terms of currents, magnetic field set-points, or similar quantities.

We refer to the set of device set-points and process variables as the *device view*, while the inputs required by the simulation codes are referred to as the *design view*. The situation is comparable to two experts from different domains working at the same table: a common understanding must be established.

We identified three patterns that together address this challenge: the *liaison manager*, the *translation service*, and the *translation object*.

Liaison Manager

The liaison manager supports mapping between the design view and the device view. It helps one actor on one side identify the corresponding actor on the other side with which communication is required. Large collaborations often employ liaison managers in an organizational sense; the software concept follows a similar idea. This pattern is part of our work from [6].

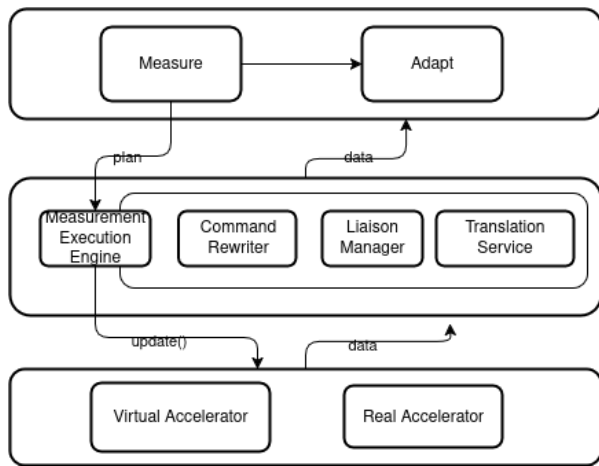


Figure 1: Layer architecture of the digital twin framework.

The liaison manager builds on the update pattern, but with a slightly different naming convention:

- In the design view, properties are identified by the tuple (elem_id, prop_id)
- In the device view, properties are identified by the tuple (dev_id, prop_id)

Note that prop_id may represent different values in the two tuples. Developers or users can query the liaison manager, for example by

```
lm.get(elem_id, prop_id) or
lm.get(dev_id, prop_id)
```

The liaison manager then responds with the corresponding tuple. A detailed description of this pattern is provided in [6].

Translation Service

While the liaison manager establishes the correspondence between the design and device views, the translation service is responsible for converting values between them. The translation service itself does not perform the conversion. Instead, it provides the user or developer with the appropriate *translation object*. This pattern is part of our work from [6].

The translation service is queried using a tuple of the form

```
ts.get([(elem_id, prop_id), (dev_id, prop_id)]).
```

The translation service will then provide the corresponding translation object. This translation object will then provide two methods:

forward for converting the value from design to device view

backward for converting the value from device to design view

The pattern is presented in full detail in paper [6].

Discussion

These three patterns provide functionality which is typically implemented (e.g. quote MML pytac?) as "unit conversion". In our mind the split up pays back as it give the functionality that unit conversion provides with a few extra:

- The split up facilitates handling e.g. the availability of hot spares. Information of hot spares can be made available within the information provided to the translation service. When the hot spare is swapped in, only liaison managers need to be informed on this change.
- The liaison manager allows users to inspect which properties of a certain device are affected by a particular element property: e.g. some simulation codes allow implementing "piggy-pack" steerers as dedicated "kick values" (e.g. in the accelerator toolbox [13]). Then the liaison manager would show that the changing this kick would correspond to a current value change of a certain steerer power converter.
- The translation service could provide different calibration curves for different translation objects. If one needs to change them — e.g. to find out if the new calibration curve would give more consistent results — one would implement a simple proxy of the translation service. It would delegate all requests to the standard translation service. Only for the ones in question it would then intercept and return another object.

We see that these patterns are also helpful for implementing Monte-Carlo simulations of e.g. commissioning (see below).

DEVICES AND MEASUREMENT PLANS

Large complexes consist of many different devices: there can be hundreds or thousands of them. Commissioning requires running dedicated tests on them. These commissioning plans are often surprisingly similar for different machines: e.g. orbit response measurement will ramp a certain current loop over one steerer after the other of any synchrotron. Only the value will be different: so we see these measurement plans similar to work instructions. Production uses "work instructions" to document manufacturing procedures so that a skilled worker can execute the foreseen production steps. Furthermore, such a work instruction would indicate to the worker which measurements should be recorded and documented for quality control.

Here we found that a measurement plan executed by an appropriate measurement execution engine would be the appropriate software implementation.

The measurement plan can be implemented as a sequence of commands or transactional commands sets. The pattern details are described in [4]. Here we shortly introduce the pattern and will discuss where we see them sensible to apply and how to apply them.

Command Execution Engine

The command execution engine receives the measurement plan. It will then handle one command after the other. Here an implementation could choose to first inspect the whole plan and see if its settings match the ranges of the involved devices. It will then only execute it if no errors are expected during execution. A more sophisticated one could test the plan on a digital twin. Only when it worked on the twin, the command execution engine would execute it on the real machine.

Measurement Execution Engine

The measurement execution engine is an enrichment of the command execution engine. It will take data along with executing the command. A rather straightforward version is to read out all devices after each step. These data should be then presented in a hierarchical data model. Another approach would be to have dedicated commands that take care which data should be read and how. This pattern is part of our work from [6].

Discussion

The measurement plan and execution engines allow separating the plan from the devices. Similar approaches were implemented by Bluesky/ophyd or ophyd-async. (These packages are part of the bluesky orchestration software stack).

The plan is then rather straightforward to serialise and save together with the measurement data. A command rewriter can be implemented based on liaison manager, translation service and translation object [6]: then a plan e.g. established in the design view can be rewritten to device view. It can then be reviewed before being executed. Furthermore it can be stored together with the data for documentation.

Similarly a command rewriter could be implemented to rewrite standard measurement plans after a facility upgrade: e.g. a new magnet could be inserted that has another relation between its field and its current: The plan could be read in and automatically adapted. Similarly command rewriters facilitate different aspects of simulated commissioning: this we will describe further down below.

Tools that need to execute measurements on the machine now only need to emit a plan to the measurement execution step. Analysis can then be performed on the returned data, which could require some post processing before analysis. Furthermore, as the plan is serialisable, it can be used for transferring a measurement plan to a remote site.

The patterns of command- or measurement-execution-engine are inspired by bluesky's run engine and ophyd device implementation [17]. We see the bluesky-queue server as a real world implementation of these patterns.

ENSURING MACHINE COMMISSIONING

Modern machines use tight tolerances; e.g. 4th generation light source using a multibent achromats have rather small dynamic apertures (i.e. the transversal area where the beam circulates around the ring in a reliable fashion)

by the design of the machine [1]. These dynamic apertures are further reduced by the variation of devices due to the manufacturing tolerance, even if produced to the state of the art. These manufacturing tolerances can be mitigated by the correction mechanisms already implemented in the machine: e.g. power converter settings can be changed to match quadrupole strengths to their design value (in particular for modern machines where each quadrupole is powered individually), steerers are used to bring the beam as close to the orbit as possible. The current state of the machine can be analysed e.g. by Linear Optics from Closed Orbits (LOCO) [19] or other procedures.

When such modern machines are designed, the lattice designer evaluates not only the quality of the ideal design. The lattice designer will also assert the quality that the machine can achieve due to the expected manufacturing tolerances of its devices and that the foreseen commissioning methods can mitigate them so that the machine will reliably achieve the required operation parameters. The quality is typically checked using Monte-Carlo methods. These sample the different set-points typically describing the random variables using normal distribution (Gaussian distribution) for the set-points. Similarly the generated data are varied.

Monte Carlo Sampling

These codes can be split up in the following components:

- the tracking part which runs on the different machine "du jour"
- the machine setup part that creates the different machines sampling the distributions for the different parameters
- the deterioration part, that adds noise to the data output of the machine
- the analysis part.

We propose the following patterns which allow reducing the cohesion of the different steps of the Monte-Carlo study (see also Fig. 2)

- describe all varying parameters as random variables with some distribution. Then the normal distribution is just a particular feature similar to the random distributions of the c++-standard library [20] or GNU scientific library [21].
- see the whole machine state (all settings of the magnet strength, magnet position, beam monitor parameters etc.) as one state that is provided by the system facade.
- When different machines are "to be produced" one can create first the "ideal machine" and then modify it using a plan consisting of a sequence of updates. A command execution engine will then take care to modify the machine. All data to be taken from this machine can be retrieved by a measurement execution engine using plans for the different measurements.

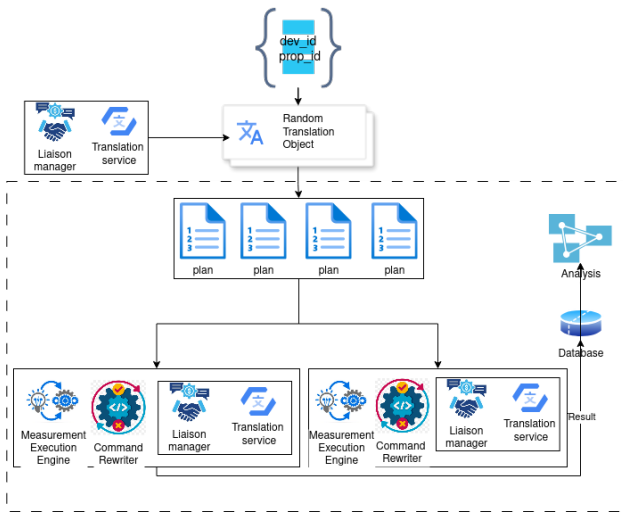


Figure 2: Implementation of a Monte-Carlo simulation environment using patterns. The dashed line indicates the part that would profit from running on a capable workflow engine.

- The data provided by the calculation engine need then to be deteriorated before returned.

Here again the liaison manager and translation service patterns can be reused. Liaison manager can redirect updates to the specific subsection, but most of the time these will be a 1-1 mapping. The translation service should then return a translation object that samples the distribution internally every time it is called. Care has to be taken that the pseudo random generator is correctly sampled by all these objects.

(Self) Adaptive Commissioning

The second part of the Monte-Carlo study targets the commissioning scripts. These scripts execute different steps, which target to measure the actual machine parameter or mitigate the effects produced by the variation of the machine parameters. Each following step can do it more precisely than the previous one but is more likely to fail if the machine is not properly tuned.

All these commissioning steps actually "adapt" or even "self" adapt the machine". Here the patterns come to help again (in particular to realise the measure part.). I refer the steps to the **Measure Analyse-Plan Execute-Knowledge** pattern [22, 23].

- every adaption step
 1. executes a measurement (the M in MAPE-K): the script needs only to create a plan. The plan is handled to a measurement execution engine for execution.
 2. analyses the measurement (the A in MAPE-K) and finds out if the result is good enough or needs further improvement
 3. if improvements are made (the P in MAPE-K) it creates a plan that will describe to which value

which parameters of the machine need to be set. These can be set as measurement plan

4. then the command execution engine takes care to change the machine.
5. if necessary one can continue the adaption at step 1.

- At the end of these steps, one will have many updates that will have modified the machine. All these steps are documented in the plan. As all of them used the update pattern. So one can use a command recorder. Here one can replay the commands if needed. Furthermore one can use the tuple (dev_id, prop_id) as keys to search for. The last value valid is the valid one if (dev_id, prop_id) refer to an absolute value. If they refer to an incremental value these have to accumulated. In that way new knowledge of the machine can be built up in a straightforward manner.

Translation service simplifies one more aspect: imagine you want to test how well the the commissioning script works even if a simplified version of the calibration curves of some magnet is used: then one can use the standard translation service for the simulation engine aspects (to map from device to design view). One then uses a proxy to the translation service which maps through all requests except for the magnet in question: here then a translation object with a simplified calibration curve is returned to the analysis script.

Practical Execution Considerations

The Monte-Carlo simulations need to run quite a number of calculations. One can see that these calculations are to a large extend workflows to execute:

- different variations of the machine need to be created
- the different commissioning steps need to be executed step by step
- analysis steps need to be executed.

The liaison manager and translation service can be used to create all the different plans for modifying the ideal machine (see Fig. 2).

It is assumed that these plans are created fast. Each plan describes what changes need to be applied to the real machine, or which noise to add to the data produced in the simulation. The commissioning steps are typically rather small scripts, which are working on the digital object of the real machine. The script execution can be delegated to a workflow engine, which would then take care of distributing the jobs to a cluster and collecting the data.

So we consider it pays back that Monte-Carlo studies blend into work-flow engines. Then all the job organisation can be handled over by tested work-flow language code: the patterns presented above simplify these steps.

TWINNING

Digital twins aim to provide a digital object that can mimic all parameters of the real machine or even more.

Switching Twin State

We see that starting a digital twin instance will typically start it as a digital object (following Kritzingers convention [24]). Here it will typically use internally the ideal object or some variation of it. Typically one wants it then to follow the state of the machine i.e. that it gets a digital shadow. Here one has to update the digital object (constantly). Finally one will want to derive new values in the twin and send it back to real machine. Here we propose a pattern to switch between the different states. Please note that one often will want to detach the digital object from the real object, e.g. to validate updates on the digital object before applying it to the machine.

So we propose a pattern that provides an consistent interface to allow switching between the different modes [4].

Implementing a Digital Shadow

The digital shadow will follow the state of the machine. This requires updating the digital objects state every time when the real state changes. Here a sniffer can be used to "sniff" the state of the machine. It will then produce a sequence of updates that need to be processed. The command execution engine is then used for updating the digital object.

Furthermore, thanks to the simple structure of the updates, one can have a filter in between, that can filter out update requests that affect the same tuple (`dev_id`, `prop_id`). This filter can then either only let pass the last one (for values that affect the absolute value) or accumulate the value (for incremental updates).

Test First on Virtual Then Run on Real

We see one key use case for digital twinning, that one wants to test or explore some settings on the twin before one does it on the real machine. Here the update patterns facilitate the whole progress.

One procedure would be: start the digital object, convert it to shadow. Then one would detach it from the real object and run some optimisations. Each optimisation step would spill out a sequence of update commands, which can be documented as command recorder.

These commands can be analysed at the end, replayed to the real object or condensed using the filter described above. simpler ones (e.g. using the filter described above)

CONCLUSION

We presented an overview of patterns for mapping design and device views, specifically for separating measurement plans from device implementations or for operating digital twins. At the core of all these patterns is the update pattern, whose central idea is that only a limited subset of parameters of the different digital objects needs to be changed and

accessed. Furthermore, it provides a straightforward and consistent interface.

Building on this design, liaison and translation services provide utilities to bridge the design and device views. The command execution and measurement execution engines then focus on separating "work instructions" from device implementations.

The patterns listed above simplify the implementation of twins or Monte-Carlo simulation environments for complex machines.

ACKNOWLEDGEMENT

We would like to thank Thomas Birke, Malte Götz, Roland Müller, Teresia Olsson, Günther Rehm, Markus Ries, Kilian Schwartz and Andreas Schällick for fruitful discussions.

REFERENCES

- [1] N. Martensson and M. Eriksson, "The saga of MAX IV, the first multi-bend achromat synchrotron light source", *Nucl. Instrum. Methods Phys. Res., Sect. A*, vol. 907, pp. 97–104, 2018. doi:10.1016/j.nima.2018.03.018
- [2] S. V. and M. Borland, "Commissioning Simulations for the APS Upgrade Lattice", in *Proc. IPAC'15*, Richmond, VA, USA, May 2015, pp. 553–555. doi:10.18429/JACoW-IPAC2015-MOPMA010
- [3] T. Hellert *et al.*, "Simulation of trajectory correction in early commissioning of the advanced light source upgrade", *J. Phys. Conf. Ser.*, vol. 1067, no. 3, p. 032001, 2018. doi:10.1088/1742-6596/1067/3/032001
- [4] W. Sulaiman Khail and P. Schnizer, "Patterns in digital twin development", in *Proc. EuroPLoP'24*, New York, NY, USA, 2024. doi:10.1145/3698322.3698325
- [5] W. Sulaiman Khail and P. Schnizer, "Building digital twins for particle accelerators: The role of architecture and patterns", submitted for Electron. Commun. EASST.
- [6] W. Sulaiman Khail and P. Schnizer, "Patterns for operating and interacting with digital twins", *Lect. Notes Comput. Sci., to be published*.
- [7] W. Sulaiman Khail and P. Schnizer, "Development of a Flexible Digital Twin Framework for Accelerators Using Design Patterns", presented at IPAC'25, Taipei, Taiwan, Jun. 2025, paper THPS050, unpublished.
- [8] J. Bengtsson, "The sextupole scheme for the swiss light source", Paul Scherrer Institute, Tech. Rep., 1997. <https://ados.web.psi.ch/slnotes/sls0997.pdf>
- [9] J. Bengtsson and M. Meddahi, "Modeling of Beam Dynamics and Comparison with Measurements for the Advanced Light Source (ALS)", in *Proc. EPAC'94*, London, UK, Jun.-Jul. 1994, pp. 1021–1024.
- [10] G. Shen, K. Shroff, and L. Yang, "NSLS-II High Level Application Infrastructure and Client API Design", in *Proc. PAC'11*, New York, NY, USA, pp. 582–584, 2011. <https://jacow.org/PAC2011/papers/MOP250.pdf>

- [11] Z. He, J. Bengtsson, M. Davidsaver, K. Fukushima, G. Shen, and M. Ikegami, “The fast linear accelerator modeling engine for FRIB online model service”, Nov. 2016, arXiv:1611.04637 [physics.acc-ph]. doi:10.48550/ARXIV.1611.04637
- [12] A. Terebilo, J. Corbett, and G. Portmann, “Accelerator Control Middle Layer”, in *Proc. PAC’03*, Portland, OR, USA, May 2003, pp. 2369–2371. <https://jacow.org/p03/papers/WPPE020.pdf>
- [13] A. Terbilo, *Accelerator toolbox for MATLAB*, SLAC National Accelerator Laboratory, CA, USA, SLAC-PUB-8732, 2001.
- [14] M. King, “High-level software frameworks at CLARA: CATAP and CLARA software tools”, presented at 2nd Accelerator Middle Layer Workshop, Berlin, Germany, Feb, 2025, unpublished. <https://events.hifis.net/event/1997/contributions/14591/>
- [15] Y. Hidaka, “Particle Accelerator MIddle LAYer (PAMILA)”, presented at Python Accelerator Middle Layer Virtual Meeting, Nov 2024, unpublished. <https://indico.esrf.fr/event/162/contributions/1042>
- [16] T. Hellert, P. Amstutz, C. Steier, and M. Venturini, “Toolkit for simulated commissioning of storage-ring light sources and application to the advanced light source upgrade accumulator”, *Phys. Rev. Accel. Beams*, vol. 22, no. 10, p. 100702, 2019. doi:10.1103/PhysRevAccelBeams.22.100702
- [17] D. Allan, T. Caswell, S. Campbell, and M. Rakitin, “Bluesky’s ahead: A multi-facility collaboration for an a la carte software project for data acquisition and management”, *Synchrotron Radiat. News*, vol. 32, no. 3, pp. 19–22, 2019. doi:10.1080/08940886.2019.1608121
- [18] T. Simko, L. Heinrich, H. Hirvonsalo, D. Kousidis, and D. Rodríguez, “REANA: A system for reusable research data analyses”, *EPJ Web Conf.*, vol. 214, p. 06034, 2019. doi:10.1051/epjconf/201921406034
- [19] J. Safranek, “Experimental determination of storage ring optics using orbit response measurements”, *Nucl. Instrum. Methods Phys. Res., Sect. A*, vol. 388, no. 1, pp. 27–36, 1997. doi:10.1016/S0168-9002(97)00309-4
- [20] *Programming Languages — C++*, International Organization for Standardization, ISO/IEC 14882:2020, Dec. 2000. <https://www.iso.org/standard/79358.html>
- [21] M. Galassi *et al.*, *GNU Scientific Library Reference Manual*, 3rd. Surrey, UK: Network Theory Ltd., 2009.
- [22] J. Kephart and D. Chess, “The vision of autonomic computing”, *Computer*, vol. 36, no. 1, pp. 41–50, 2003. doi:10.1109/MC.2003.1160055
- [23] D. G. D. L. Iglesia and D. Weyns, “MAPE-K formal templates to rigorously design behaviors for self-adaptive systems”, *ACM Trans. Auton. Adapt. Syst.*, vol. 10, no. 3, 2015. doi:10.1145/2724719
- [24] W. Kritzing, M. Karner, G. Traar, J. Henjes, and W. Sihn, “Digital twin in manufacturing: A categorical literature review and classification”, *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1016–1022, 2018. doi:10.1016/j.ifacol.2018.08.474