PROSHELL – THE MEDAUSTRON ACCELERATOR CONTROL PROCEDURE FRAMEWORK

R. Moser, A. B. Brett, M. Marchhart, C. Torcato de Matos, EBG MedAustron, Wr.Neustadt, Austria
J. Gutleber, CERN, Geneva, Switzerland
J. Dedič, S. Sah, Cosylab, Ljubljana, Slovenia

Abstract

MedAustron is a centre for ion-therapy and research in currently under construction in Austria. It features a synchrotron particle accelerator for proton and carbon-ion beams. This paper presents the architecture and concepts for implementing a procedure framework called ProShell. Procedures to automate high level control and analysis tasks for commissioning and during operation modelled with Petri-Nets and user code is implemented with C#. It must be possible to execute procedures and monitor their execution progress remotely. Procedures include starting up devices and subsystems in a controlled manner, configuring, operating O(1000) devices and tuning their operational settings using iterative optimization algorithms. Device interfaces must be extensible to accommodate yet unanticipated functionalities. The framework implements a template for procedure specific graphical interfaces to access device specific information such as monitoring data. Procedures interact with physical devices through adapter software components that implement one of the following interfaces: (1) state-less or (2) state-driven device interface. Components can extend these device interfaces following an object-oriented single inheritance scheme to provide augmented, device-specific interfaces. As only two basic device interfaces need to be defined at an early project stage, devices can be integrated gradually as commissioning progresses. We present the architecture and design of ProShell and explain the programming model by giving the simple example of the ion source spectrum analysis procedure.

INTRODUCTION

MedAustron [1] [2] is an ion therapy and research centre presently under construction in Wiener Neustadt, Austria. The facility features a synchrotron-based accelerator (Figure 1) with up to 5 ion sources for protons, carbon ions and possibly other light ions. It will provide ion beams with energies up to 800MeV to 5 beam lines, one of which is a rotating proton gantry.

The Procedure Shell Execution Framework (ProShell) is a C# application to automate high level control and analysis tasks for commissioning and during operation. Each task called a procedure implements a standardized procedure interface and is deployed as .NET assembly (shared objects). Key features of the ProShell are:

- Allocating resources on behalf of a procedure.
- Uniform access to system, software and physical devices independent of communication protocols for monitoring and control purposes.
- Reception and visualization of device measurements
- Management of generic procedure lifecycle and custom procedure workflow.
- Parallel execution of multiple procedures
- Automatic procedure execution without user intervention
- Manual procedure execution to step through the procedure specific workflow.
- Provide access to control system services hiding implementation specific interfaces and communication protocols.

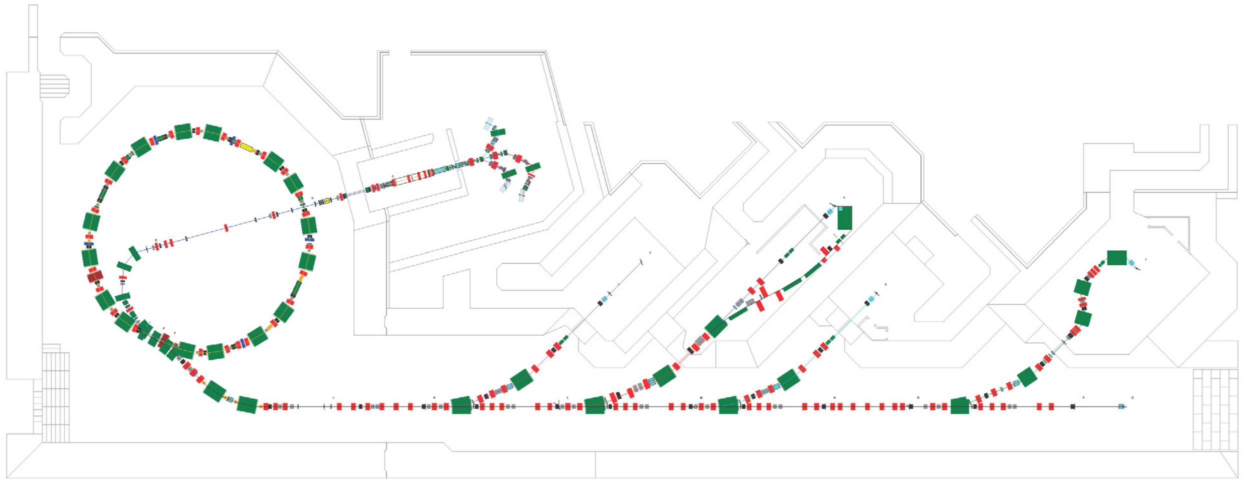

Figure 1: MedAustron accelerator layout.

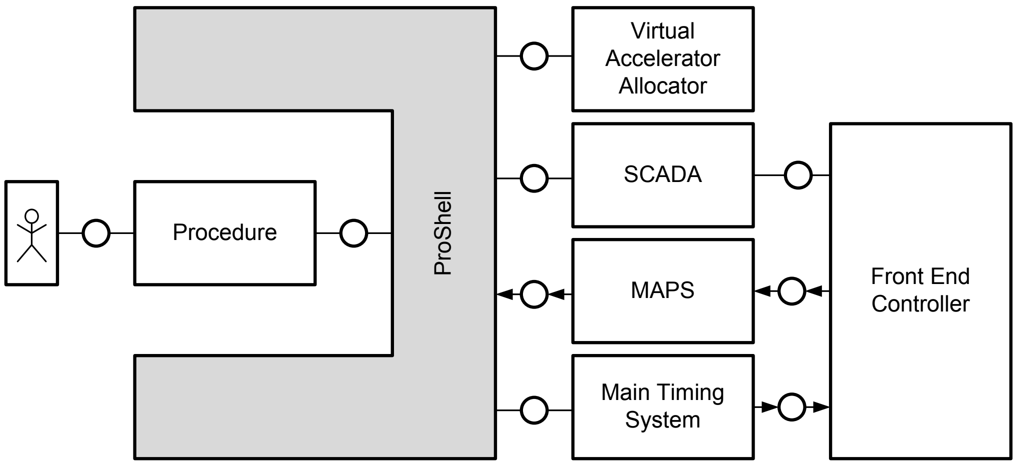

Figure 2: General ProShell architecture and main communication partners.

ARCHITECTURE

Overview

ProShell is a framework to dynamically load and execute procedures implemented as C# classes. As outlined in Figure 2 it provides access to system, software and physical devices for monitoring and control purposes. These services are accessible from ProShell through service-specific *Driver* objects that are used internally and are not directly accessible from the loaded procedures:

- *Virtual Accelerator Allocator* (VAA) is the scheduler of the system that allocates resources for exclusive usage on behalf of a user application.
- *WinCC OA* is a Supervisory Control and Data Acquisition (SCADA) tool from Siemens [5]. It acts as the main communication backbone between user interfaces and procedures on tier 1 and frontend controllers and devices on tier 3 [7].
- *MAPS services* are a set of data servers implementing a publisher subscriber protocol. It forwards measurements and main timing information from front-end controllers to user applications in non-real-time.
- *Main Timing System* (MTS) generates events for beam generation that are delivered to the frontend controllers with a precision of 100ns [4].

In MedAustron the last three services provide a communication channel to the *frontend controllers* running on National instruments PXI crates. Each frontend controller can host multiple *frontend devices* that implement device specific tasks in LabView to control the connected physical device. They also provide a unified interface to WinCC OA through a shared variable OPC server and may emit fast monitoring data through MAPS.

Procedure

A *Procedure* encapsulates specific repetitive control and processing tasks for operation and commissioning written in C#. It is compiled into a .NET assembly (shared library) and implements a common interface to be managed by the ProcedureContext independent of its task.

Procedures may call any available C# functions and libraries. However, procedures should mainly use the API provided by the ProcedureContext to interact with system devices and services.

Procedure Context

ProShell encapsules execution of procedure instances in separate *ProcedureContexts*. Each ProcedureContext acts as a container that provides coordinated access to devices and control system services and manages the procedure lifecycle. Thus it provides the capability to execute procedures in parallel. In case of resource conflicts the VAA will delay the allocation of one procedure and subsequently ProShell will delay the execution of this procedure. As depicted in Figure 3 each ProcedureContext manages the following objects separately:

- *Procedure* implements a specific control or processing task and provides a standardized interface to be controlled by the ProcedureContext.
- *ProcedureLifecycle* handles the general lifecycle of the procedure that is common to all procedures.
- *PetriNetEngine* is a workflow engine that executes the procedure specific workflow defined in a Petri Net Modelling Language (PNML) file [3].
- *DeviceCache* allocates resources on behalf of the procedure using the VAA driver and keeps a cache of C# adapter objects for communicating with the resources.

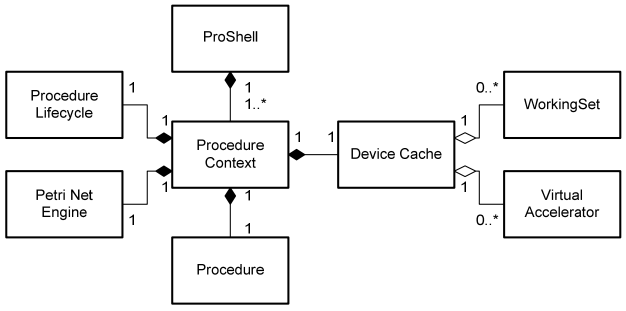

Figure 3: Class diagram for object owned by the ProcedureContext.

Resources

All device instances and types are represented in WinCC OA as data points (DP) and data point types (DPT). In addition each DPT contains a set of data point elements (DPE) that are name-value pairs with a defined value type. Each device implements one of the following interfaces:

- *BasicDevice* is a state-less front-end device interface that only provides a minimal set of DPEs for monitoring.
- *StateDrivenDevice (SDD)* is a state-driven front-end device interface that extends the BasicDevice with additional DPEs to provide an interface for commanding and login to guarantee exclusive access to a device.

WinCC OA also provides two special resource types to control a set of devices concurrently through a single virtual device:

- *Working Set* (WS) is a virtual device that implements the SDD interface and controls a set of SDDs.
- *Virtual Accelerator* (VAcc) controls a set of Working Sets and subsequently a set of devices. It also implements the SDD interface. In addition a VAcc also contains a dynamically assigned Main Timing Generator that allows a procedure to emit timing information for beam generation with an accuracy of 100ns.

Resource Adapters

ProShell encapsulates communication over a number of different communication technologies, shielding procedures from the underlying addressing and communication specifics by providing devices as object that follow the adapter pattern [6].

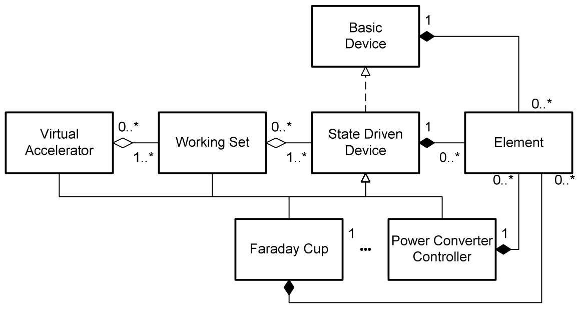

Figure 4: Class diagram of resources adapters for devices, Working Sets and Virtual Accelerators.

Adapter objects provide an object-oriented API that implement BasicDevice or StateDrivenDevice interfaces as shown in Figure 4. Each device exposes a number of elements and functions:

- *Elements* wrap a single name-value pair (i.e. DPE) independent of communication protocol using a specific Driver internally. Classes implementing the element interface provide data type conversion for complex data types and client-side validity checks not supported by WinCC OA.
- *Functions* provide access to multiple Elements that require a specific workflow when reading or writing. In this case the Elements will be available only as protected fields within the device adapter.

General Procedure Lifecycle

The *ProcedureLifecycle* manages the lifecycle part, which is the same for all procedures following the state machine depicted in Figure 5.

When a procedure is opened a new ProcedureContext is created, the procedure specific PNML file is loaded and the state is moved to *Hold*. At this time no resources are allocated.

Subsequently the user can issue the *Initialize* transition to request the allocation of the specified resources from the VAA. The procedure lifecycle state moves to *Ready* when all resources have been allocated.

Thereafter the user may emit an *Enable* transition that parameterizes the procedure specific Petri net and moves the state machine into *Op* state. While in *Op* state the procedure specific Petri net can be executed in steps or run until termination.

The *Disable* transition stops the execution of the PetriNet and the *Finalize* transition releases all allocated resources. If an error is detected, the state moves to *Failed* and a subsequent *Clear* transition stops the petri net and releases all resources.

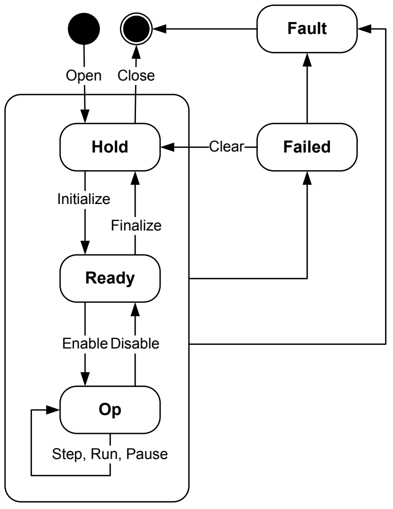

Figure 5: Procedure Context Lifecycle.

Petri Net

The Petri net for each procedure is specified in a Petri net modelling language (PNML) file [3]. The implemented petri net engine extends the standard petri net defined in PNML with the following functionalities:

- Support *coloured tokens* to pass data between transitions.
- Attaching *callback functions* to transitions to execute procedure specific code.
- Extension for asynchronous execution by putting tokens into places programmatically during runtime. This is symbolized in the PNML with asynchronous arcs (dashed lines).

BEAM SPECTRUM ANALYSIS

The ion source beam spectrum analysis procedure detects the particles types generated by a specific ion source. Therefore a current is applied to the bending magnet and the generated field deflects the generated particles depending on the particle mass. The energy of the particles that hit the following faraday cup is measured. Due to the correlation of current to particle type, the generated particles can be detected with peak detection algorithm on the energy over current plot.

```
public override void OnEnable()
{
 _faradayCup.Move(true, 20);
 PetriNet net = Context.PetriNet;
 net.BindParameter("n", (uint)Currents.Cur.Count);
 IPlace place = net.GetPlace("conf");
 foreach (Tuple<double> current in Currents.Cur)
 {
   CurrentToken token = new CurrentToken()
   { Current = current.Item1 };
   place.AddInitialToken(token);
 }
 Context.PetriNet.Trigger();
}
```

Figure 6: *Enable* transition for beam spectrum analysis.

This procedure will allocate an ion source working set containing the power converter for the bending magnet and the following faraday cup during the *Initialize* transition. The subsequent *Enable* transition as depicted in Figure 6 moves the faraday cup into the beam line and configured the *conf* place with n current tokens.

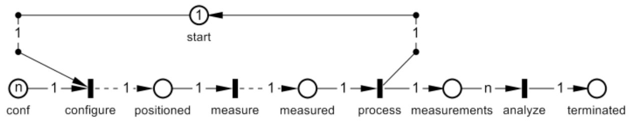

Figure 7: Ion source beam spectrum measurement with synchronous arcs (solid lines) and asynchronous arcs (dashed lines).

After the procedure was brought into *Op* state, the Petri net depicted in Figure 7 becomes active. The Petri net engine will remove a token from *conf* and *start* places and fire the *configure* transition (Figure 8) that applies the current specified in the token from the *conf* place. When the set-point was reached a token will be put into the *positioned* place and subsequently the *measure* transition is fired. During this transition the faraday cup is instructed to perform a measurement that will be put asynchronously as a token into the *measured* place.

```
private void Configure(PetriNet petrinet,
  ITransition transition,
  Dictionary<IArc, Queue<IToken>> input,
  Dictionary<IArc, Queue<IToken>> output)
{
 List<CurrentToken> tokens =
  petrinet.GetTokens<CurrentToken>(input);
 if (tokens.Count == 1)
 {
  var cycleToken = (CurrentToken) token;
  _pcc.CurrentAqn.Subscribe(
   petrinet.GetPlace("positioned"), token[0].Current);
  _pcc.CurrentCcv.Value = token[0].Current;
  petrinet.GenerateTokens(output);
 }
}
```

Figure 8: Petri net callback function for *conf* transition.

After a measurement has been performed for each current token in the *conf* place, n tokens with measurement results will be available in the *measurement* place and the *analyze* transition will be fired. This transition will plot the measurements and performs a peak detection to detect the generated particles.

SUMMARY

This article presented the architecture and selected architecture significant components of the MedAustron Procedure Shell Execution Framework. The application follows a container-based approach where each procedure is executed in its own sandbox. Accelerator devices are controlled and monitored through resource adapter objects that provide an object oriented API to hide communication specific details.

Integration with the main systems (WinCC OA, MTS, VAA and MAPS) has been concluded. First tests have been carried out in the MedAustron test column with the presented procedure and generic procedures that execute beam cycles using physical devices where available and emulating not-available devices through WinCC OA scripts. These tests have shown that generic tasks such as allocation, data conversion and validity checks can be encapsulated in ProShell to provide a simplified interface for procedures and that the Petri net provides valuable feedback for the currently running procedure.

Next steps for ProShell include extending element classes for additional protocols, and specifying and implementing adapter objects and procedures required for commissioning and during operations.

REFERENCES

[1] M. Benedikt, A. Wrulich, "MedAustron—Project overview and status", Eur. Phys. J. Plus (2011) 126: 69.

[2] M. Benedikt, A. Fabich, "MedAustron—Austrian hadron therapy centre", Nuclear Science Symposium Conference Record 2008. NSS '08. IEEE, pp.5597-5599, 19-25 Oct. 2008.

[3] J. Billington et al., "The Petri Net Markup Language: Concepts, Technology, and Tools", Proc. 24th Int. Conf. Application and Theory of Petri Nets (ICATPN'2003), Eindhoven, The Netherlands, June 2003.

[4] J. Dedic et al., "Timing System for MedAustron Based on Off-The-Shelf MRF Transport Layer", in Proc. IPAC 2011.

[5] P. Golonka, M. Gonzales-Berges, "Integrated Access Control for PVSS-Based SCADA Systems at CERN", in Proc. ICALEPCS 2009.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns—Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[7] J. Gutleber et al., "The MedAustron Accelerator Control System", Proc. ICALEPCS, 2011.