EPICS 7 at ITER

Migrating ITER CODAC Supervision to using PVXS

B. Bauvir (CD/CCI)
W. Van Herck (CD/DCS)

Disclaimer: The views and opinions expressed herein do not necessarily reflect those of the ITER Organization

Outline

- Context
- Rationale for adopting EPICS 7 in CODAC SUP
- Objectives and architecture
- Findings
- Achieved software quality
- Conclusions and forecast

Context

- ITER Control, Data Access and Communication (CODAC) Supervision (SUP) software components
  - SUP is a distributed system with multiple components (automation, configuration, monitoring and timing), deployed at all levels of the control system hierarchy (from the top central level all the way to the individual plant system I&Cs)
  - SUP provides integrated monitoring and control to machine operators and acts as their main interface with the ITER machine
- ITER has committed to building its Instrumentation and Control System (I&C) using EPICS and its eco-system

Rationale

- EPICS 7 (pvAccess) adoption
  - Executing User-supplied code as part of data processing workflows
    - Remote Procedure Call (RPC) with support for discoverable User-defined datatypes
  - Loading Plant System configuration
    - Adherence to defined Quality of Service criteria
      - Atomic and synchronous load operation
      - Ensuring data integrity through to remote controllers
      - Providing exception handling and reporting mechanism
  - Supporting device profiles and reconstructing software models
    - Pressure measurement in ITER Cooling Water System has 81 *independent* IOC records

EPICS 7 in CODAC SUP

- EPICS Collaboration Meeting Spring 2019 contributions

  Interfacing MARTe2 to the EPICS Channel Access and pvAccess protocols

  ITER plant configuration system
- SUP
  - C++ with pvData, pvDatabase, and pvAccess (client and RPC)
  - Python with pvapy and bindings atop SUP C++ RPC classes
  - Matlab code encapsulation with C-API atop SUP C++ RPC classes and datatype support by means of JSON serialisation
- MARTe2
  - C++ with pvData, pvDatabase, and pvAccess
  - .. not affected by the porting exercise

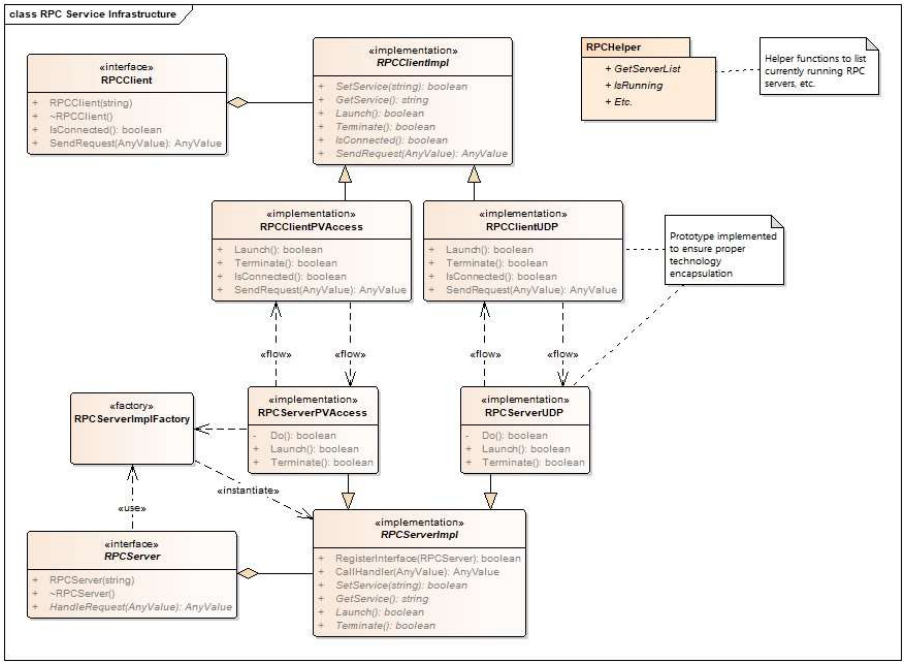
Findings (2019)

- Complicated APIs
  - pvData datatype declaration and reflection,
  - Accessing attributes, manipulating arrays, etc.
- Multiple implementations available and quite terse documentation
  - Concerns about evolutions, maintainability, interoperability
    - ‘epics::pvAccess::RPCClient’ vs ‘epics::pvaClient::PvaClientRPC’
- Support for arrays of structure generally missing in tools
- And more recently
  - Robustness issues during the tear-down process
  - Interoperability between pvDatabase and pvaClient

Software architecture

- Objective
  - Homogeneous handling of structured variables in application code
    - Identical application-side API, whether Plant Operation Network, or Real-time Data Network (SDN), or ..
    - Implies own datatype support
  - Strong technology encapsulation
    - Systematic use of bridge pattern
    - Aiming at binary backward compatibility

Software architecture



Migration

- Precautions have been in place from the onset of adopting EPICS 7 in SUP to facilitate change of technology
  - Limited impact of porting .. restricted to SUP C++ implementation classes
  - Even the unit tests were limited at accessing interface classes
- Implemented first a set of GTest atop the PVXS API
  - Focussing on required patterns, and
  - ‘pvxs::Value’ and ‘pvxs::server(::SharedPV)’ as these are the most distant from pvData and pvDatabase used in past versions

The Good ..

- Datatype declaration

```
pvxs::Value value;

bool ret = !static_cast<bool>(value);

try
  {
    value = pvxs::TypeDef(
      pvxs::TypeCode::Struct, "pvxs::test::MyStruct_t", {
        pvxs::members::Struct("header", "pvxs::test::MyHeader_t", {
            pvxs::members::UInt64("counter"),
            pvxs::members::UInt64("timestamp"),
          }),
        pvxs::members::Float64("value"),
        pvxs::members::UInt8A("padding"),
      //pvxs::members::StructA("array", {
        pvxs::members::StructA("array", "pvxs::test::MyStructArray_t", {
          pvxs::members::UInt8("one"),
          pvxs::members::UInt8("two"),
        }),
      }).create();
  }
catch (const std::exception& e)
```

The Good ..

- Parsing and accessing pvxs::Value fields ..

```
template<typename T> Value&
    pvxs::Value::operator=(const T& val);
pvxs::Value operator[](const std::string& name);
```

```
value["header.timestamp"] = ::ccs::HelperTools::GetCurrentTime();
value["value"] = 0.1;
value["array[1].one"] = 1;
value["array[2].two"] = 2;
```

The Good ..

- Lambdas (and the 15SLOC RPC server)

```
__pv = pvxs::server::SharedPV(pvxs::server::SharedPV::buildMailbox());
__server->addPV(this->GetService(), __pv);

// Install callback .. as lambda expression
__pv.onRPC([this](pvxs::server::SharedPV& pv, std::unique_ptr<pvxs::server::ExecOp>&& op, pvxs::Value&& request)
  {

    ccs::types::AnyValue _request;

    bool status = ccs::HelperTools::SerialiseAnyValueFromPVXSValue(_request, request);

    ccs::types::AnyValue _reply;
    pvxs::Value reply;

    if (status)
      {
        log_debug("PVAccessRPCServer::[]() - Calling registered handler ..");
        _reply = CallHandler(_request);
        log_debug("PVAccessRPCServer::[]() - .. returned");
        status = ccs::HelperTools::SerialiseAnyValueToPVXSValue(_reply, reply);
      }

    if (status)
      {
        op->reply(reply);
      }

    return;

  });
```

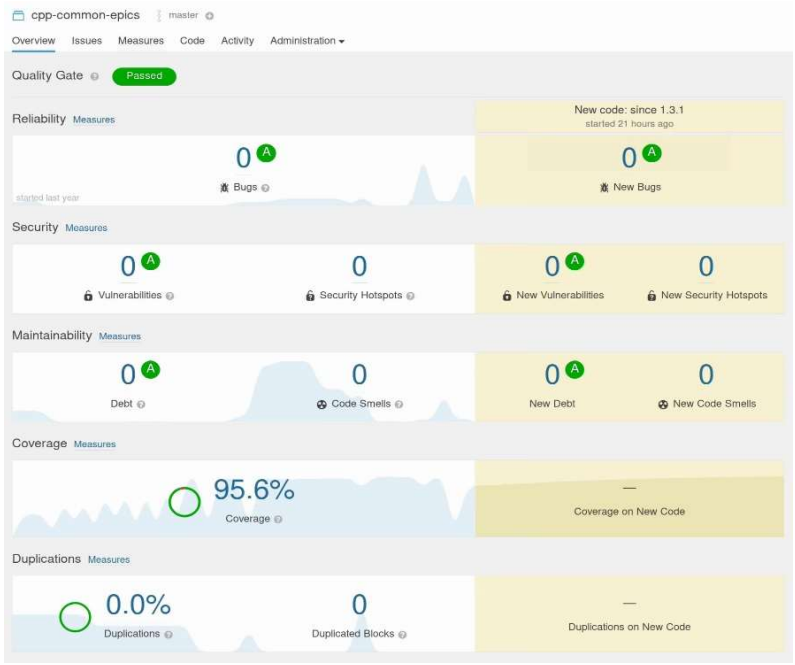
Not systematically exploited due to limitations of static analyser in use

The less good ..

- Handling of arrays still is cumbersome
  - Initialisation, access .. yet very similar to pvData so not a porting issue .. just an issue with the porter

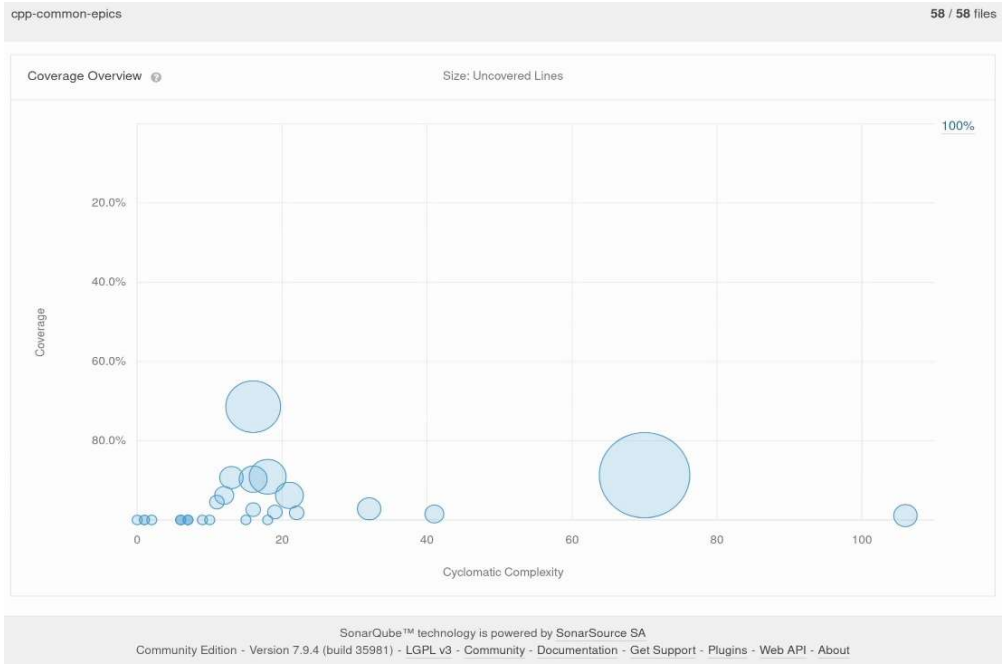
```
bool status = static_cast<bool>(pv_value);

if (status)
  {
    pvxs::shared_array<uint8_t> _array (size);

    for (::ccs::types::uint32 index = 0u; (status && (index < size)); index++)
      {
        _array[index] = ::ccs::HelperTools::GetElementValue<::ccs::types::char8>(&in_value, index);
      }

    pv_value = _array.freeze().castTo<const void>();
  }
```

Resulting software quality metrics



Based on PVXS snapshot made on 26/06/2020

Resulting software quality metrics

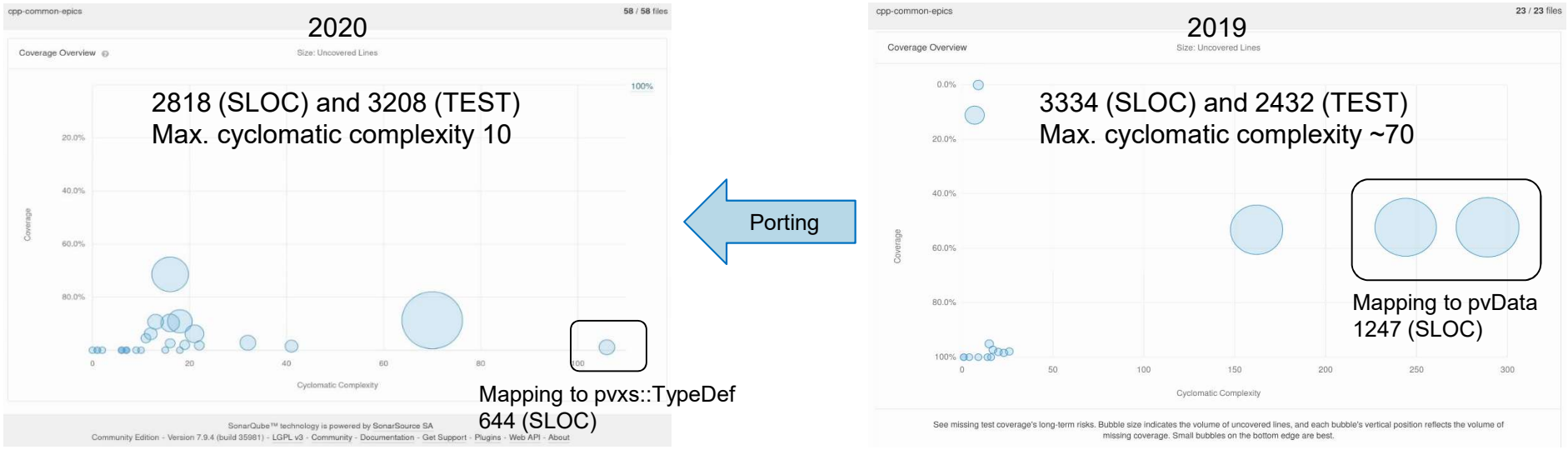


Based on PVXS snapshot made on 26/06/2020

Resulting software quality metrics

- Reduced code size and complexity
  - Higher coverage achieved with maintaining GTest code untouched ..

  .. perceived lowered coverage is an artefact of the smaller code size



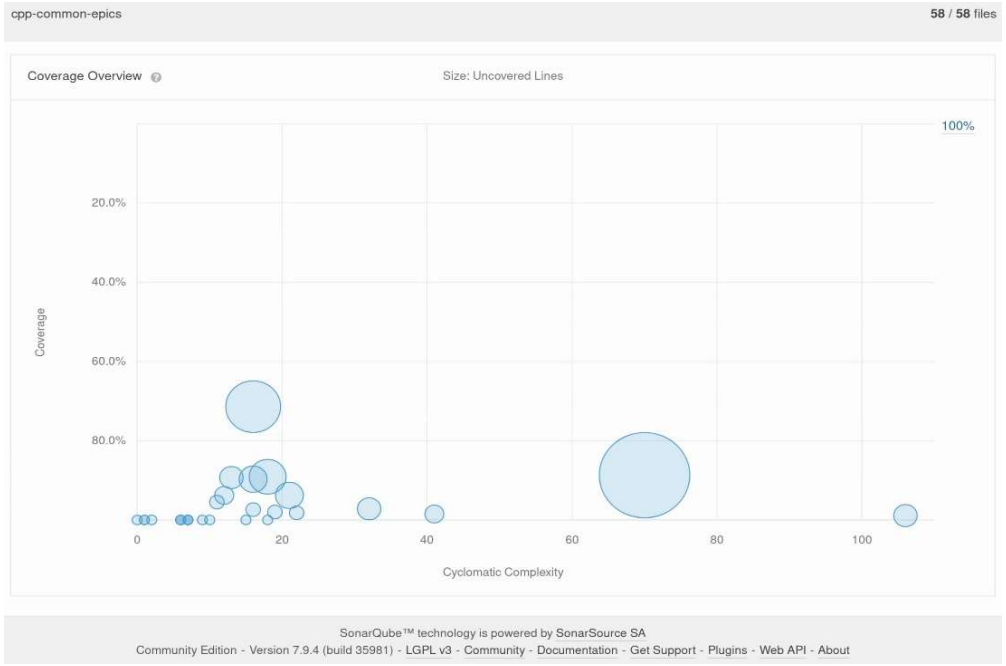
Conclusions

- Porting from pvDataCPP and pvAccessCPP
  - Quite straightforward after familiarisation exercise in the form of implementation of bare PVXS API automated tests
    - 4 man-days overall
- Facilitated thanks to the systematic use of bridge pattern
  - I would have abandoned if I had made the mistake to use plain pvDataCPP and pvAccessCPP constructs throughout the corresponding application code
- Improved quality and maintainability through lower complexity and smaller code base

Forecast

- Continue to extend GTest suite dedicated to PVXS API and features ahead of PVXS v1 release and integration in CODAC Core System
  - Include automated tests as part of our Continuous Integration (CI) infrastructure
    - Achieve notification in case of future changes breaking compatibility, operability, etc.
  - Build confidence in adopting PVXS as full part of high-integrity CODAC software
- Re-evaluate EPICS 7 support in tools, CS-Studio, etc.
  - Unclear if issues reported in 2019 have progressed

Thank you for your attention

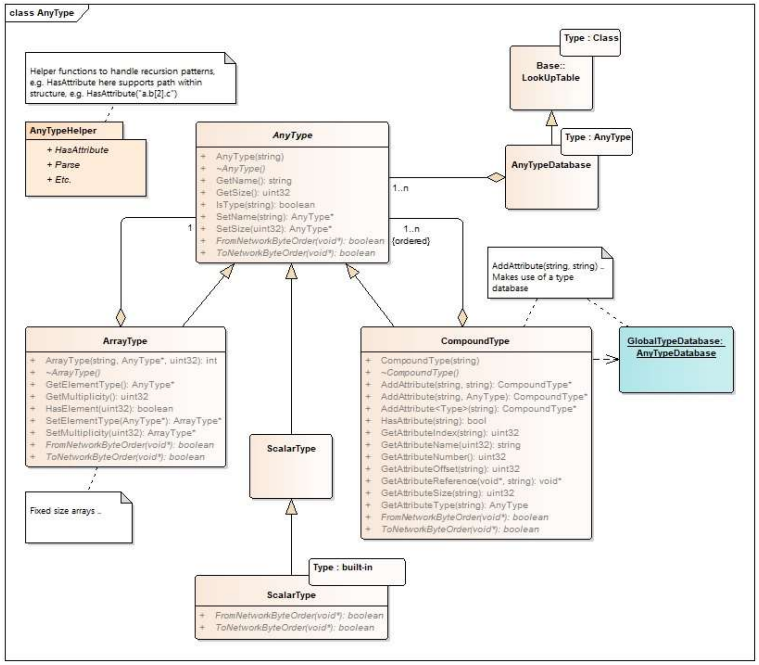


Back-up slides

Architecture goals

- Containment of complexity
  - Provide abstraction mechanisms, manageable information and control means
- Scalability
  - Support integration of new components with limited impact on existing ones
- Changeability
  - Adapt to changing requirements with affordable resources
- Robustness
  - Prevent propagation of failures through the control system

Software architecture



Software architecture