ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG004

ROBUST AND MAINTAINABLE OPERATION USING BEHAVIOR TREE SEMANTICS

W. Van Herck†, R. Gunion, R. Lange, G. Pospelov
ITER Organization, Saint-Paul-lez-Durance, France

Abstract

The ITER machine's inherent complexity and the diverse operational phases, such as commissioning and engineering operations, present significant challenges in balancing operability, integration, and automation.

The open-source software framework *oac-tree* was designed to create, maintain, and execute operational procedures in a robust and repeatable manner. Its semantics are based on a behavior tree model, which inherently supports reactive behavior, making it well-suited for goal-oriented tasks.

To accommodate diverse use cases — from small-scale tests to fully integrated operations — the framework's architecture was built with composability and extensibility in mind. System-specific interfaces and user interactions are fully decoupled from the core library, ensuring flexibility and adaptability.

The oac-tree library is deployed in production at ITER, offering a command-line interface for executing procedures as system daemons or for interactive use. Its maintainability is ensured by a minimal, low-complexity codebase with well-encapsulated third-party dependencies.

The oac-tree ecosystem also includes a Qt-based graphical user interface and a server enabling multiple clients to interact with running procedures. A plugin mechanism supports framework extensions, with existing plugins available for EPICS-based control systems, mathematical expression evaluation, common control system logic, and ITER-specific network protocols.

INTRODUCTION

During the different phases of the ITER machine, many operational procedures need to be defined, verified and executed in a traceable and maintainable way.

For high-level operations, these procedures roughly fall into one of the following categories:

- Monitoring: low level signals are aggregated into higher level statuses that can then be consumed by central control applications.
- Control: high-level goals are expressed as specific combinations of low-level states or goals. This allows applications to reuse those control procedures, avoid duplication of detailed implementations and facilitate their maintenance.
- Automation: this typically combines both monitoring and control to be able to react to changing machine state in the most appropriate way.

Regardless of their category, these procedures will need to evolve during the lifetime of ITER for several reasons:

- Change of environment: as more plant systems are being added, their presence may require adapting monitoring and control logic.
- Increased automation: manual procedures can be replaced by automated ones as they mature and their interactions with the ITER plant are better understood.
- Improved performance: better understanding will also lead to opportunities to fine-tune procedures to maximize operational efficiency.

Furthermore, during integration and commissioning of plant systems, procedures will need to be executed to carry out specific tasks in a repeatable and traceable way. Those procedures are likely to be adapted even more often.

Oac-tree [1] is a software tool meant to facilitate the creation, adaptation, approval and execution of operational procedures. It also defines a text-based format for these procedures, allowing version control and easy traceability. It provides a means to replace procedural documents with unambiguous instructions that can be executed by the tool.

Using extensions that implement the ITER protocols for plant system configuration and monitoring, oac-tree can be effectively used as the main tool for central control and automation of the ITER plant. With proper extensions, it can just as easily be deployed in other environments, such as during commissioning of plant systems as mentioned above.

GOALS AND REQUIREMENTS

One of the main goals for the ITER control system is to maximize operational efficiency, i.e. to reduce the time spent on tasks that do not directly contribute to the scientific mission of the machine.

Automation of manual procedures is a critical step towards this goal, but care has to be taken that the benefits are not offset by increased costs, due to maintenance and manual interventions.

While high-level programming languages, like C++, Java or Python, provide almost unlimited flexibility, they come at a high cost in terms of maintenance, since small code changes can introduce unexpected instabilities that may be hard to detect with unit testing. Hence, the use of these languages at the level of operational procedures would require extensive test plans, often involving complex test environments and considerable human resources.

Software frameworks that expose some higher-level semantics, like finite state machines, solve this problem by having a more constrained interface that can be tested more easily during automated tests. This trade-off between flexibility and robustness is quite common and finding the right balance between these can be challenging and requires clear understanding about the functional requirements of the framework within its operational context.

† walter.vanherck@iter.org

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG004

Many operational tasks, from low-level commissioning procedures to high-level automated control and coordination tasks, need to exhibit reactive behavior, i.e. the ability to switch between different sets of actions based on certain machine conditions. Without explicit support for such reactivity, a software automation framework will be limited in its application and will not be able to replace most manual tasks by automated ones.

Control systems rarely operate in isolation; they must interface with a variety of external systems, protocols, and user interfaces. Integration challenges arise from differences in data formats, communication standards, and timing requirements. The architecture must provide well-defined interfaces and abstraction layers to decouple the core logic from system-specific details, enabling seamless interaction with plant equipment, supervisory systems, and operator consoles. Successful integration also requires extensibility, so that new components and protocols can be incorporated without major redesigns. Ideally, extended functionality can be added to the framework without changes of the basic code. This adheres to the open-closed principle in software engineering and prevents new functionality from breaking existing functionality. User input/output needs to be also supported to allow operators to interact and influence running procedures, for example to assert that some manual actions have been successfully performed.

It is possible to achieve flexibility without compromising robustness by using composition of elementary and well-tested building blocks. Composability can also prevent duplication of parts of procedures as mature subprocedures can be reused in different high-level procedures.

Finally, due to regulatory compliance, a certain level of quality assurance is necessary. The structure or semantics of the chosen software architecture may not be directly determined by such quality requirements, but will have a big impact on the amount of resources needed to achieve the required quality level. Very modular frameworks with simple interface definitions are much easier to test and will thus require less time to achieve compliance with a given set of quality criteria.

FRAMEWORK OVERVIEW

Behavior Trees

High level representations of control flows such as finite state machines and behavior trees are often used in control system architectures. For the oac-tree framework, behavior trees were chosen as the main concept for its architecture. After a brief overview of behavior trees, the justification for this choice will be presented in terms of its advantages over other possible choices.

Behavior trees [2] form a hierarchical model for representing complex sequences of operations and decision logic. Instructions are modeled as nodes within a tree structure, allowing for flexible composition of behaviors such as sequencing, selection, parallel execution, and conditional branching. Each node encapsulates a specific action, condition or control flow, and the tree is always traversed starting from the root node. This is done by ticking the root node, which will then propagate the ticks down according to its specific control flow logic.

One of the strengths of behavior trees is the simplicity of the interface of an instruction node. When these nodes are ticked, they return a success, failure or running state, where 'running' means that the node is asynchronously executing some action. So, while the ticks propagate downwards, i.e. from parent to child nodes, the statuses that are returned propagate upwards in the tree hierarchy. This interface facilitates the implementation of custom nodes in a robust way, as no complex logic needs to be exposed over its interface. It also allows to easily test the behavior of single nodes, as one only needs to check the correct return status under different conditions.

One of the main advantages of behavior trees is that they naturally support reactive behavior. Figure 1 shows a simple example of how a small behavior tree can support reactive switching of actions based on a condition.

In the figure, the root node represents a reactive sequence, which will execute its second child node (the normal operation path) as long as its first child returns a success status. The first child is a reactive fallback, that will only execute its second child if the first child fails. So, as long as the condition (system is OK) is satisfied, the fallback node will report success, and the root node will continue executing the normal operation path. The root node will continuously check the status of the first child node while the normal operation path is executing. Whenever the system's state changes and the condition is no longer satisfied, the fallback node will start executing the failure path and the root node will interrupt the normal execution path.

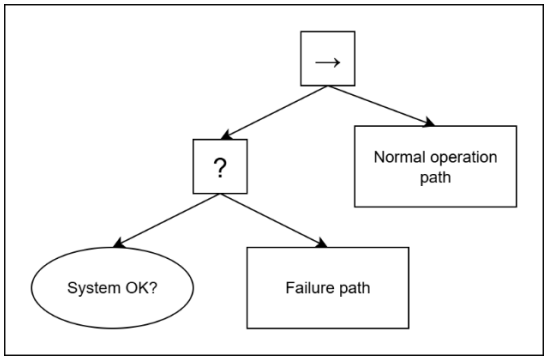

Figure 1: Reactive behavior.

Behavior trees offer several advantages compared to hierarchical finite state machines (HFSMs). Unlike HFSMs, which can become complex and rigid as the number of states and transitions grows, behavior trees provide a modular and composable structure. This allows developers to build complex behaviors by combining simple, reusable nodes, making the system easier to extend and maintain.

Behavior trees naturally support parallelism, conditional execution, and dynamic decision-making, which are often cumbersome to implement in HFSMs. Their hierarchical organization enables clear visualization of control flows and simplifies debugging. As was noted in [2], behavior

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG004

trees implement two-way control transfer, since each instruction node is ticked by its parent. In contrast, in HFSMs there is no way to directly deduce the previous state the system was in.

Overall, behavior trees enhance scalability, maintainability, and adaptability in control system architectures, making them well-suited for complex and evolving automation tasks.

The framework's behavior tree implementation supports advanced features such as breakpoints, monitoring, and asynchronous input handling. Nodes can be decorated with additional logic (e.g., decorators for repetitions or inversion), and the execution status of each node is tracked to provide robust error handling and reporting.

Architecture and Concepts

The core elements of the behavior tree implementation of oac-tree are instruction nodes and variables. Procedures are composed of a tree of instruction nodes and a single workspace, containing variables that can be accessed by all instruction nodes.

Instruction nodes can be classified as:

- Control flow nodes and decorators: these nodes have child nodes (exactly one for a decorator) and will tick these child nodes according to a specific logic. These are the basic building blocks to implement control logic.
- Action nodes: these nodes will perform a specific action and return the result of that action as success or failure. Most action nodes will be performing these actions asynchronously, i.e. they return a 'running' status while the action is not yet completed. This allows their parent nodes to check certain conditions that may require the action to be interrupted. An example of such behavior was shown above.
- Condition nodes: as for action nodes, these nodes do not have child nodes. However, they will never return a 'running' status and can be seen as immediate evaluations of a condition, where the result is reported to the parent node (success/failure).

Variables in the workspace implement getters and setters for their values, which can be any structured data. This data is defined by a custom data transfer object, which can hold primitive scalars (signed/unsigned integers of different sizes, floating point values, booleans and strings) or structures and arrays of arbitrary depth. Subfields of this data in a variable can be accessed by addressing it with its field name (for structures) or index (for arrays).

This generic interface allows different implementations of variables to hold their data in memory, on disk or remotely accessible through a network protocol.

Both synchronous and asynchronous behavior is supported in oac-tree. It contains sequence and fallback instructions with memory, which means they do not reevaluate previous child nodes at each tick. Figure 2 shows an example where this can be very helpful. Suppose a specific task consists of the execution of three subtasks in a given order. If we were using a normal, i.e. reactive, sequence, then the execution of 'Async task 1' would cause the others to be interrupted at each tick of the parent node. Using a sequence with memory, the behavior is now as one would expect: it runs the child tasks in order and will finish when either one of the child tasks fails or all have succeeded.

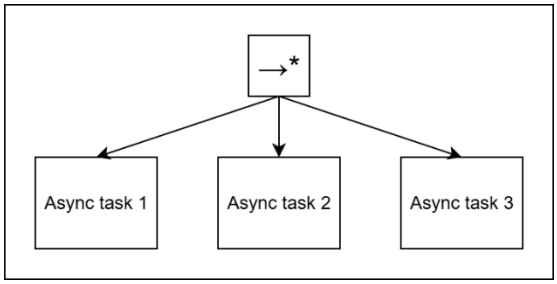

Figure 2: Fixed steps of a subtask.

Another example can be seen in Figs. 3 and 4. Here the goal is to detect if a system is OK by inspecting a fixed number of signals.

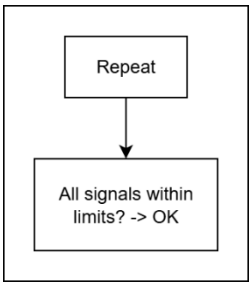

Figure 3: Asynchronous listening to signals.

In the asynchronous implementation (Fig. 3), there is a busy loop over the action. The action itself will determine the system status (OK/Not OK) from the current values of the signals and write it to a variable. Even if the signals rarely change value, this small procedure will consume a significant amount of CPU resources.

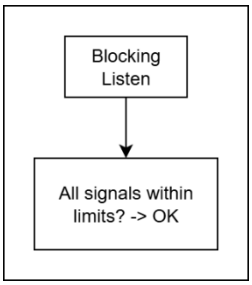

Figure 4: Synchronous listening to signals.

In the synchronous case (Fig. 4), there is a blocking **Listen** instruction that will only tick its child when it is notified of an update of any of the signals it is listening to. When that happens, the system's status is again calculated and published to a variable. Because there is no busy loop and the **Listen** instruction blocks as long as there is no signal update, this procedure will barely consume CPU resources when the updates are rare.

Monitoring procedures typically benefit from synchronous implementations, while control procedures depend heavily on reactive and asynchronous behavior.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG004

COMPOSABILITY AND EXTENSIBILITY

Modular Design

While behavior trees inherently support composability, oac-tree provides some additional features to facilitate the composition of larger trees from smaller ones.

Special control flow nodes were developed to segregate the definition of reusable subtrees from the main control tree.

The first one is the **Include** instruction node. Prior to execution, this node will be replaced with the full subtree it refers to, which can be located in the same or even a different procedure file. This instruction avoids duplication of instruction tree definitions and allows to create procedure files that contain a predefined and tested set of subtrees that can be consumed by higher-level procedures. This is reminiscent of how programming libraries expose users to well-tested subroutines for a specific domain.

Next, there is also an **IncludeProcedure** instruction that not only will be substituted with a full subtree, but will also have access to a separate workspace of variables. In this way, variables that are specific to those subtrees can be hidden from the main procedure, leading to improved readability and better separation of concerns. This is similar to how most programming languages provide a scope for variables.

Extensions

As a generic framework for control procedures, the main oac-tree library is agnostic of the concrete (network) interfaces of ITER's diverse plant systems. However, the software supports extension of its functionality by a plugin mechanism. Oac-tree plugins can define specific instruction nodes and new types of workspace variables that hide machine-specific interfaces behind the generic interface for instructions/variables.

Extensibility is achieved through a robust plugin mechanism. The framework is designed to load and integrate plugins at runtime, enabling third-party developers to add new instructions or variable types without altering the main codebase. Plugins are registered via a central registry, which manages their lifecycle and ensures compatibility with the core system.

Since the ITER control system relies heavily on EPICS interfaces [3] with the different plant system I&Cs, a plugin was developed to integrate EPICS based systems with the oac-tree framework. The EPICS plugin provides:

- Read/write instructions for EPICS process variables (Channel Access/PvAccess).
- An instruction to perform a Remote Procedure Call (RPC) with an EPICS PvAccess RPC Server.
- Client variables that monitor process variables (both Channel Access and PvAccess).
- A variable that hosts a PvAccess process variable, i.e. a server variable.

Other plugins were also developed to facilitate common tasks and to interface with ITER-specific network protocols:

- Mathematical expression plugin for the evaluation of mathematical expressions.
- Strings plugin for the manipulation of string variables.
- System plugin for executing shell commands and accessing the system clock.
- Control plugin providing common control logic functions, such as waiting for a condition to become true with a timeout.
- Supervision and automation system (SUP) plugin: this plugin provides instructions for plant system configuration, pulse counter management and timing functions.

Each plugin adheres to well-defined interfaces, ensuring that it can participate in the behavior tree execution and interact with other components. This approach also encourages community contributions.

USER INTERFACES

A key architectural goal is the decoupling of the user interface from the core execution logic. The framework defines clear boundaries between the backend (behavior tree execution, job management, variable handling) and the frontend (user interaction, visualization). Communication between these layers is facilitated through well-defined interfaces and message-passing mechanisms.

This separation allows for multiple user interface implementations (CLI, daemon, graphical UI) to coexist and interact with the same backend logic. The backend exposes APIs for querying job status, submitting user input, and receiving notifications, while the frontend is responsible for rendering information and collecting user actions. This design promotes maintainability, testability, and the ability to evolve the user interface independently of the core system.

The next subsections will briefly describe the currently existing user interfaces in the oac-tree framework.

Command Line Interface

Oac-tree procedures can be executed from the command line by using either the **oac-tree-cli** or **oac-tree-daemon** executable. The main difference is that the daemon implementation is designed to run in the background and does not support user input instructions.

Oac-tree Server

A server application was implemented that manages a set of procedures and exposes a specific network interface to start, stop, pause and resume these procedures. The GUI can act as a client to interact with such servers over the network.

The network protocol used between the server and its clients is a custom protocol based on EPICS.

Graphical User Interface

The oac-tree graphical user interface (GUI) comprises the main oac-tree desktop application, which serves as an IDE for behavior trees (**oac-tree-gui**), and several auxiliary tools, including a structured-data editor (**anyvalue-editor**), a process variable (PV) monitor (**sup-pvmonitor**), and a procedure execution monitor (**oac-tree-operation**).

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG004

A typical workflow looks like this: the user imports one or more oac-tree XML files, extends them in the node editor or instruction tree view, adds network and local variables, and then validates behavior directly in the GUI using breakpoints and step-by-step execution. When finished, the project is exported back to oac-tree XML. Later, procedures can be executed via the oac-tree remote server, while the GUI can attach to a running procedure to observe and control its execution.

The GUI is implemented in C++ and built on the Qt framework. Architecturally, it follows the Model–View–ViewModel (MVVM) pattern [4] to separate business and presentation logic from the user interface:

- Model: The "application model" encapsulates data (such as instruction and variable descriptions) and the rules for operating on it (e.g., which properties are editable in a given context).
- ViewModel: Prepares data for the UI, enforces validation and editing workflows, and exposes observable state and commands.
- View: Thin Qt widgets that render the prepared data and route user intent; they contain minimal logic.

Applying the MVVM pattern consistently across the application enables broad test coverage without launching the main window.

The GUI is intentionally decoupled from the core oac-tree classes. Communication with the domain is limited to well-defined points - import/export and job monitoring - so most GUI code never touches domain types. At the same time, the GUI remains aware of domain changes (new instructions/variables, attribute schema updates) via the oac-tree plugin mechanism. This allows the GUI and the oac-tree library to evolve independently at a different pace, following different reasons to change, and with minimal friction.

DEPLOYMENT AT ITER

The following subsections list some use cases where oac-tree was successfully deployed in production at ITER.

Monitoring

To provide a common way to interpret plant system state during a pulse, Common Operating States (COS) were defined as a finite state machine that needs to be exposed for each single plant system I&C.

For multiple plant systems, this state machine is implemented as an oac-tree monitoring procedure. These procedures subscribe to changes in plant system specific state variables and calculate and publish the COS state that corresponds to those state values.

Control

The startup and shutdown procedures for the different cooling water loops have been implemented as oac-tree control procedures and are actively used in production as a way to improve robustness, avoid human errors and let operators focus on tasks with high added value.

These procedures are highly hierarchical and are composed of multiple procedure files. At the top-level, the startup procedures contain a sequence of steps that need to be followed to startup the pumps of a cooling water loop:

- Verify the pre-requisites for starting up the loop.
- Setup the pressurizer (clear interlocks, set level and pressure control).
- Prepare the pump startup (selection of pumps, valve control, etc.)
- Start the pumps (start the pumps, set flow control, etc.)

To be able to react to unforeseen issues during the pump startup, the procedure uses a fallback node. This allows to execute an alternative sequence of instructions in case something goes wrong during the nominal execution path.

These procedures also contain instructions that interact directly with the operator, for example to get confirmation to proceed or to retrieve configuration settings.

Automation

While ITER is not yet performing pulsed operation, prototype procedures have already been prepared to coordinate all the required actions that are needed to run a pulse.

In particular, a prototype automation procedure was written to execute the following steps:

- Ensure all (required) plant systems are in the right state to be configured.
- Validate plant system configurations and configure the plant systems according to the goals of the current pulse.
- Initiate a countdown, during which the plant systems will initialize themselves in accordance with their configuration.
- Start the execution phase of the pulse, in which central control is handed over to the real-time system for plasma control.
- Wait for the plasma control system to indicate it has ended control and take back control at this point.
- Perform post-pulse checks and bring all systems back into their initial states for the next pulse.

Fallback behavior is implemented to react to failure conditions during any of these phases.

CODE OVERVIEW AND QUALITY

Overview

Oac-tree is fully implemented in C++ (C++17 standard, with public headers conforming to C++11). It is targeted for Linux systems, although users have been able to successfully compile and use it on MacOS.

The basic library consists of about 13k lines of C++ code (excluding tests), of which 5k for the basic instruction set and 500 for the basic variables. This means that only 7.5k lines of code are in the interface definitions, parsing and execution engine.

Due to the simplicity of the instruction and variable API, most plugins have between 500 and 1000 lines of code.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG004

Dependency Management

Third-party dependencies are minimal and well encapsulated to prevent the need for major code changes when these are being updated.

Internally, dependency injection is used where it benefits decoupling of separate components and facilitates unit testing. As mentioned in the section on user interfaces, this is what enables the procedure execution engine to interact with a variety of user interfaces.

Code Quality

The framework demonstrates a strong commitment to code quality through several practices:

- Modular Organization: The codebase is organized into logical modules (attributes, base utilities, instructions, parser, procedure, execution engine, variables), each with clear responsibilities and minimal coupling.
- Interface-Driven Design: Abstract interfaces and base classes are used extensively, promoting extensibility and reducing duplication.
- Error Handling: Exception classes and status tracking mechanisms are implemented to ensure robust error reporting and recovery.
- Testing: Dedicated test directories and scripts (including unit and integration tests) are provided to validate functionality and prevent regressions.
- Documentation: There is comprehensive documentation, covering API usage, core concepts, and extension guidelines.

Due to regulatory compliance, the oac-tree library and the plugins that are used in production need to be compliant to our highest software quality level, which implies:

- No major issues (or worse): Parasoft [5] is used to perform static code analysis, using the MISRA C++ 2023 rules.
- Unit test coverage > 95 %.

The unit tests in this framework demonstrate a strong commitment to code coverage and validation. The code contains a comprehensive suite of test files, each targeting specific components and functionalities of the library, such as instruction handling, variable management, error handling, and utility functions. These tests are designed to exercise both typical and edge-case behaviors, ensuring that the core logic is robust and reliable.

In the base library, there are more than 2500 expect/assert statements in about 500 unit tests, with a code coverage > 95 %.

For the GUI, all major user operations— import/export, copy/paste, undo/redo, drag-and-drop, and more — are covered by unit and integration tests, with over 15,000 expect/assert/on_call statements across the suite.

FUTURE PROSPECTS

The majority of the code for the base library and the ITER-provided plugins is in maintenance mode, meaning that they receive only minor quality improvements and bug fixes.

However, the GUI is still under active development. As more user feedback is gathered, improvements for the user experience (UX) will be gradually incorporated.

Active development is currently also underway to provide a plugin that interfaces with OPC UA.

Finally, as ITER is expected to host many different procedures for monitoring and control (on the order of 100s), we plan to use container orchestration tools to deploy these procedures. This will facilitate multiple aspects regarding their management: start/stop services, monitoring, providing redundancy, etc.

CONCLUSION

This paper presents oac-tree, an open-source framework used at ITER to author, validate, and execute operational procedures in a robust, repeatable way. Motivated by the complexity of ITER's operations across commissioning, engineering, and future pulsed operation, the framework adopts behavior trees to achieve reactive, goal-oriented control while balancing flexibility, maintainability, and traceability. The paper argues that behavior trees offer a simple, testable interface (success/failure/running), natural support for reactivity and parallelism, and clearer composability than hierarchical finite state machines.

Architecturally, oac-tree centers on instruction nodes (control-flow, action, condition) and a shared workspace of structured variables. It supports both asynchronous and synchronous patterns, enabling efficient monitoring and responsive control. The framework extends composability through Include and IncludeProcedure, promoting reuse and separation of concerns. Extensibility is provided via a plugin mechanism that adds instruction and variable types without changing core code. Existing plugins integrate EPICS, provide common utilities (math expressions, strings, system calls, control logic) and support ITER's supervision/automation needs (SUP), with an OPC UA plugin in development.

User interfaces are deliberately decoupled from the execution engine through well-defined APIs and message-based communication. Procedures can be run via a CLI or daemon, orchestrated by a server exposing a custom EPICS-based protocol, and created/debugged with a Qt-based GUI. The GUI includes tools for structured data, PV monitoring, and live procedure observation, and supports stepwise execution and breakpoints. Broad unit and integration testing underpin the tooling.

Deployed use cases at ITER include monitoring (computing Common Operating States across plant systems), control (robust startup/shutdown of cooling water loops with operator interaction and fallback handling), and prototype pulse automation (pre-pulse validation/configuration, countdown, execution handoff to real-time control, and post-pulse recovery). The codebase (C++17, Linux-targeted) emphasizes minimal dependencies, dependency injection, and a compact core (~13 k LOC excluding tests). Quality is enforced via MISRA C++ 2023 static analysis (Parasoft) and > 95 % unit test coverage. Future work focuses on GUI UX improvements, an OPC UA plugin, and

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEAG004

containerized orchestration to manage the large number of procedures expected in production.

DISCLAIMER

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

REFERENCES

[1] Oac-tree, https://github.com/oac-tree

[2] M. Colledanchise and P. Ögren, *Behavior trees in robotics and AI: an introduction*, Boca Raton, FL: CRC Press, 2018.

[3] EPICS Controls, https://epics-controls.org

[4] MVVM pattern, https://learn.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps

[5] Parasoft, https://www.parasoft.com