ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THMPL03

A NEW SIMULATION ARCHITECTURE FOR IMPROVING SOFTWARE RELIABILITY IN COLLIDER-ACCELERATOR CONTROL SYSTEMS[*]

Y. Gao[†], T. Robertazzi, Stony Brook University, Stony Brook, NY 11794, USA
R. H. Olsen, J. T. Morris, K. A. Brown, Brookhaven National Laboratory, Upton, NY 11973, USA

Abstract

The control systems of the Collider-Accelerator Department (C-AD) at Brookhaven National Laboratory (BNL) are complex systems consisting of approximately 1.5 million [1] control points. Instances of C-AD control systems are applied in the Linear Accelerator (Linac), Electron Beam Ion Source (EBIS), Tandem Van de Graff pre-accelerators, the Booster accelerator, Alternating Gradient Synchrotron (AGS), and the Relativistic Heavy Ion Collider (RHIC). Its performance has a crucial impact over the whole accelerator suite. In this paper, we propose a new simulation framework that can improve the robustness of the control system. It focuses on enhancing the reliability of its software codes by running automated testing. The architecture is described, followed by some key use cases in the current system. Moreover, the next development phase is proposed.

INTRODUCTION

The Relativistic Heavy Ion Collider (RHIC) is a world-class particle accelerator at BNL. It enables [2] scientists to study what the universe may have looked like in the first few moments after its creation. RHIC contains two 3.8 kilometers counter-rotating super-conducting rings to carry particle beams which can be collided in six crossing regions [3,4] to provide possible interactions for experimenters to study.

The RHIC control system provides [5–8] the operational interface to the collider and injection beam lines. The architecture is hierarchical and consists of two physical layers with network connections: Console Level Computers (CLCs) level and Front-end Computers (FECs) level, as seen in Fig. 1. The front-end level comprises more than 500 FECs, running the VxWorks^{TM} real-time operation system. Each FEC consists of a VME chassis with a single-board computer, network connection, and I/O modules. The FECs are distributed around 38 locations, including the control center, service buildings and 18 equipment alcoves accessible only via the ring tunnel. Along with [3,4] data links and hardware modules, they are the control systems interface to physical accelerator equipment. The console level is the upper layer of the control system hierarchy, which consists of operator consoles, physicist workstations and server processors that provide shared file, database and general computing resources. Processes known as managers are also found at the console level. A manager can function as a sort of virtual FEC. Some managers perform data concentration or processing functions. Increasingly, CLC level manager

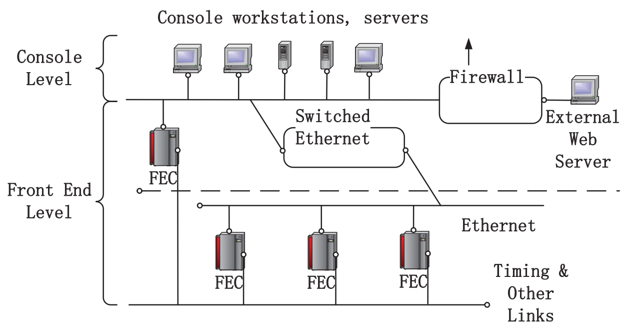

Figure 1: RHIC system hardware architecture.

processes bypass the FEC level and provide a direct interface to accelerator equipment. CLC managers communicate with accelerator equipment via direct network connections or via Ethernet-to-GPIB (General Purpose Interface Bus) or Ethernet-to-serial converters. These CLC level equipment interface managers have been chosen as the first target for our simulation architecture.

The software system for RHIC controls is structured with [5–8] well-defined layers and interfaces following a standard object-oriented design paradigm. At the front end level, it mainly consists of two broad categories: device drivers and Accelerator Device Objects (ADOs). Device drivers used in RHIC FECs are very similar to device drivers used in other applications in that they provide a standard interface for software to interact with various hardware devices. An ADO is a software object which serves as a collection of various related accelerator control points and provides standard methods through which higher-level applications can interact with those parameters.

Reliable functioning of control systems is critical to the proper operation of accelerators. In this paper, we propose a new simulation architecture for C-AD control systems, which will enable automated testing of controls software. The new simulation platform analyzes ADO code and generates a corresponding test bench. Test results are shown to users for codes analysis. Key use cases and future plans of the simulation platform will be discussed.

PRELIMINARIES

In this section, we describe the main components of the C-AD control systems for the new simulation framework.

Accelerator Device Object

The Accelerator Device Object (ADO) is the fundamental construct [4,9] in C-AD control systems. The ADO model is a flexible way to view accelerator equipment. It was intro-

[*] Work supported by Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy.
[†] ygao@bnl.gov

duced during the development of the RHIC Control System in the late 1990s.

ADOs are instances of C++ or Java classes which abstract features from underlying controls hardware into a collection of collider control points known as parameters, and each parameter can possess one or more properties to better describe characteristics of devices. The number of parameters and names of parameters are determined by ADO designers to meet the needs of the system. The most important ADO class methods for device control are the *set()* and *get()* methods. The *set()* and *get()* methods are processed by the ADO that acts as the interface to device drivers to access controls hardware. The collider is controlled by users or applications which *set()* and *get()* the parameters in instances of these classes using a suite of interface routines.

Since the crafting of ADOs is so crucial, special development tools are used. The source file of an ADO is typically stored in a file with a ".rad" extension, which stands for RHIC ADO Definition file. A preprocessor was written to transform ".rad" file source codes into C++. It takes care of the necessary details and allows ADO designers to focus on the controls interfaces, which are the *set()* and *get()* methods and event codes. One of the primary goals of the ADO concept is to establish unified standards for controls software development, which automates the integration of device level controls into the overall control systems and simplifies the coding process.

System Components and Tools

Controls Name Server The Controls Name Server (CNS) provides [10] a centralized repository where unique name/value pairs can be efficiently managed and queried. In C-AD control systems, applications can get enough information from CNS through its object's name entries so that the associated data can be accessed.

Notification Server The Notification Server (notifServer) [11–13] in C-AD control systems is a server that receives notifications from RHIC ADOs. It then logs the notices in a daily log and forwards them to the AGS Alarm Receiver. The notifications can be displayed by alarm display applications.

Logging System The C-AD control system provides a logging service [14], which is used to save machine parameters and device values to provide a history of the accelerator performance and to provide data to be analyzed for machine physics studies. Data collection and retrieval by the logging system is carried out by the base tools [15] called the SDDS Toolkit, developed at Argonne National Laboratory (ANL). SDDS Data is stored in a directory tree structure. Separate applications for displaying logged data have also been provided.

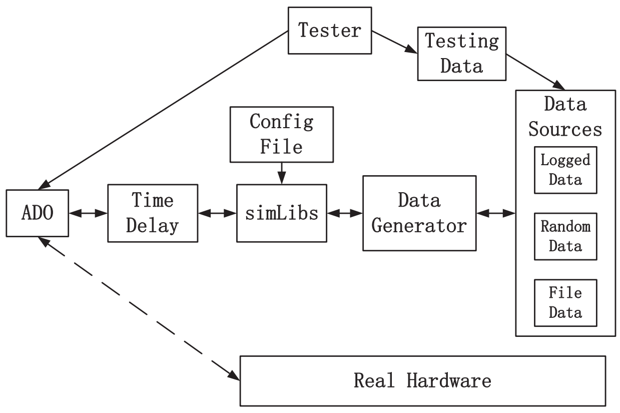

Figure 2: Simulation architecture overview.

SIMULATION ARCHITECTURE OVERVIEW

In this section, we describe the new proposed simulation architecture, which is shown in Fig. 2. It focuses on running test data to improve an ADO's reliability. The architecture includes software modules that simulate interactions between ADOs and hardware.

One of the fundamental goals of the simulation architecture design is to enable control system developers to easily switch between normal mode and simulation mode. In the normal mode (lower branch in Fig. 2), an ADO is communicating with real hardware. In the simulation mode, an ADO is interacting with the simulation platform (upper branch in Fig. 2). We call ADOs that use simulation mode "simulated ADOs". When developers want to switch from the normal working mode to the simulation mode, they set an environment variable to specify the directory which contains the simulated library. Thus the simulated library will be used at compile time instead of the real library. No changes are made to the ADO code itself. Unset the environment variable to go back to the normal working mode.

Simulated Bus Interface and Data Generator

The two basic blocks in the simulation structure are "simLibs" and "dataGenerator". They work collectively to carry out the main testing procedure.

In real cases, communication between ADOs and devices takes place over Ethernet and the particular bus interface connecting to the hardware, such as General Purpose Interface Bus (GPIB, IEEE 488.2). Under simulation mode, the "simLibs" block acts as both the Ethernet and special bus interface, connecting ADOs to the data generator block. It parses commands from ADOs, and responds with corresponding test data, which are generated from the data generator. Practically, the simLibs block contains all the variable and function declarations in the original GPIB header file, and rewires the connections of the functions to customized test data instead of real hardware.

Test data can come from three sources:

Logged Data Logged data are history data logged using the methods described in the previous section. Reading logged data saves time for arranging data in the right format for particular ADOs. By using logged data, developers can replay a period that they are interested in, e.g. a 3-hour simulation run of the time before a magnet failure could help to figure out the cause.

Random Data Random data are generated for special testing purposes based on ADO parameters' requirements (e.g. random integers, random doubles, random strings, etc.). A useful case would be generating random data to cover the entire data range of a parameter to see if the ADO codes can handle it.

File Data File data are test data written by developers. File data are organized with one-to-one pairs, with the first element storing a keyword and the second element storing the corresponding test data. During a simulation, based on the commands received from ADOs the simLibs block loads the test file and looks for particular keywords and fetch the corresponding test data to interact with ADOs correctly. Thus file test data is good for situations in which the developer has an understanding of the conditions that tend to cause ADO failures.

To get an idea of how those two blocks can work together to replace real device, consider the following example. Suppose an ADO is querying current value of a magnet by issuing a device command "$MEAS : CURR?$". In the normal working mode, upon receiving this command the underlying device will respond with a current value. Whereas in the simulation mode, instead of the real hardware the simLibs block receives this command, parses and passes it to the data generator, and returns with different responses depending on the testing mode (log, random or file) set by the developer. In the log mode, data response is drawn from history data of the device. Which time period of the history to use is specified by developers in the configuration file. In the random mode, data response could be a random double number within a range. The range is based on the hardware specification and is specified by developers in the configuration file. In the file mode, the file will contain a one-to-one pair, for example, "$current\&97.98$", where "&" is the delimiter used to separate the file keyword and the test data. In the configuration file, developers need to specify another one-to-one pair "$MEAS : CURR? \rightarrow current$" to map[1] the command "$MEAS : CURR?$" to the file keyword "$current$", so that the simLibs block can find the right entry in the file and respond with the developer-specified test data "97.98".

Generalization of Bus Interface

One of the essential guidelines in designing the simulation platform for ADO testing is the generalization of the testing process. In real cases, ADOs often communicate with different hardware, who usually have different sets of device commands, and use different kinds of bus interfaces. How to accommodate various situations is a crucial step. In real operations, hardware information is provided and processed by the hardware itself. The bus interface does not need to understand hardware information, and it merely provides connections to enable communication. However, in the simulation mode, since there is no real hardware involved, a counterpart is needed to deal with hardware information in order to generalize the testing process.

The simulation environment makes use of a configuration file to address this concern. The Simulated Hardware Configuration File uses standard XML [16] format. It consists of a list of information that contains all necessary information needed to interact with a particular type of hardware. It is also easily maintained by appending or removing the corresponding information sets. Once simulation begins, the "simLibs" module loads the configuration file and then is aware of all the hardware going to be used in this simulation. Therefore it understands how to respond to various commands it receives from ADOs.

We supply a Java GUI to assist in the creation of the Simulated Hardware Configuration File. There are several levels in the structure of the configuration file. The top level lists all ADO classes representing different types of devices that are going to be simulated. Inside each device section, there are several parts describing the simulation parameters. The first part is the initialization part, which initializes the GPIB parameters, including indicator of whether the device is connected to the network, whether the device is turned on, and whether developers want to print out debug information, etc. The following parts are the testing mode parts (default mode, log mode, random mode, file mode) describing the parameters used in each testing mode[2]. Those parts share a common feature that they all consist of a group of basic blocks, each of them is a one-to-one pair mapping from a device command to the corresponding response information. Depending on the testing mode, response information varies. For example, if the ADO issues a "$MEAS : CURR?$" command to get the current value of a magnet. In the log mode, response information is mainly related to the instructions about how to find the logged data in a log file. The configuration file could contain entry "$y6 - q89 - master : currentM$", where "$y6 - q89 - master$" is a system-wide generic name used by CNS to locate the device from which the ADO is querying information, and "$currentM$" is the device parameter which holds the magnet's current value. Another entry "$Number; 0; 0$" to indicate the parameter value is a numerical type, and the remaining part "0; 0" are the indexes to determine the position in the log file to fetch data. Every time the simLibs gets a data point, it increases the index so that the history data can be replayed in real time. In the random mode, response information is about how to generate random data (e.g. random array data, random integers, random strings, etc.). An example entry could be "90.12 100.23" to

[1] The pairs are written in standard XML format, here is just an illustration.

[2] If the developer does not specify testing mode, default mode will be used, in which case all hardware responses in the simulation are predefined constant values.

indicate that the return data is generated randomly uniformly between 90.12 and 100.23. Whereas in the file mode, response information is a keyword which will be used to find the actual response data in a testing file, which is written by the developer before the simulation starts. An entry[3] in the configuration file could be "$MEAS : CURR? \rightarrow current$", where "$current$" is the file keyword used by the simLibs to get the corresponding data in the test file. Moreover, in the log mode part, developers also need to specify the time period of the history from which the log data is drawn, and the directory of the log file.

Time Delay Module

To better simulate real world communication scenarios, a time delay module is added between the simulated library and ADOs. The default case is no delay, i.e. an ADO gets responses immediately after commands are issued. However, a developer can require a delay added in the following way. First, the developer specifies 3 environment variables to indicate the values of a probability p, a potential delay value d and a binary $flag$. For example, $p = 0.5, d = 10$, then after the simulation starts, the communication between ADO and the simulated library will experience delay with probability 0.5, otherwise there is no delay. Specifically, the simLibs block first generates a random decimal number, if that number is smaller than 0.5, then the simLibs block adds a time delay before it responds to any command from the ADO. Depending on the $flag$ value, the actual delay value added will be generated uniformly randomly between 0 and 10 seconds if $flag = 1$, or the 10 seconds will be used directly if $flag = 0$. Moreover, based on the actual delay experienced, timeout errors handling is also implemented in the simLibs block. Hence whenever the ADO waits for a response longer than a predefined timeout value after it sends out the command, the simLibs block will issue a timeout error and set the corresponding error bits to indicate the status.

Tester

The block on the top of Fig. 2 is used to automatically generate test files for the file testing mode as mentioned above. "Tester" goes through ADO codes, and finds out information about parameters in the ADOs, such as number of parameters and type of parameters. Next, according to that information and the developer-specified Tester running mode, it generates test data and stores it in text files. One text file is created for each ADO class. Those text files are used as input test files when simulation begins.

In order to run the Tester block, there are two Tester configuration files developers need to create. The first file lists all ADO classes developers want to test in the simulation. Each class starts with a "#" as the delimiter. Following each ADO class, developers need to specify how to run the "Tester" to generate the testing files. Available Tester running modes are "MAX" or "MIN", which stands for giving parameters of the ADO values that are larger than the maximum threshold or smaller than the minimum threshold based on the ADO specifications. The second file consists of a group of one-to-one pairs, with the first element of each pair listing the parameter name of the ADO and the second element being a keyword to be used later to write a test file. For example, if the developer wants to test the parameters of an ADO named "bopglPS" by crashing the parameters' upper limits, then the first file should contain entries "#$bopglPS$" followed in the second line "$bopglPS\&Mode\&MAX$". The second file lists pairs, for example, "$currentM\&current$" and "$voltageM\&voltage$", where "$currentM$" and "$voltageM$" are the parameters "$bopglPS$" uses to monitor the current and voltage values of a magnet, "$current$" and "$voltage$" are file keywords. After the Tester block starts, it loads in the first file to find ADO class names, then connects to the database using the class names ("$bopglPS$" in this case) to locate the ADOs and get all of their parameter ("$currentM$" and "$voltageM$" in this case) information (number, type, engineering limits, etc. as mentioned above). Next, based on the Tester running mode ("MAX" in this case) the Tester randomly generates test data[4], and writes the data into a test file using the keywords ("$current$" and "$voltage$" in this case) listed in the second file. In this example, the test file generated will contain entries "$current$&a big number" and "$voltage$&another big number". Later this file will be used as test data source to verify if ADO codes can handle those cases.

Testing Procedure

Before the simulation starts, the first thing to do is to set up environment variables, which includes setting the running mode as simulation mode (instead of using the real hardware), specifying testing mode (can be random mode, log mode, file mode, or default mode[5] if left unset), the directory to load the configuration file, and the time delay parameters (p, d and $flag$).

The next step is to create the Simulated Hardware Configuration File. As mentioned above, the configuration file is written in standard XML format and contains device information for the simLibs block to interact with the ADOs. The file is created by a Java GUI, in which developers are asked to specify a location to save the file[6].

After the Simulated Hardware Configuration File is created, developers can start the simulation and switch testing modes during run time by changing the value of the environment variable to "LOG", "$RANDOM$" or "$FILE$".

The simulation results are the values that are published in the parameters of the simulated ADO. They can be viewed or logged using standard control system tools. Depending on the results, users can verify the functionality of ADO

[3] See 1.

[4] For the "MAX" mode, test data are in the range of $(10 - 100)\times$ the maximum threshold; For the "MIN" mode, test data are in the range of $(10 - 100)\times$ (the minimum threshold -1).

[5] See 2.

[6] This directory should be consistent with the directory specified by the setup environment variable.

codes. Future versions of the simulation architecture may include specialized mechanisms for capturing and comparing simulation results.

Key Use Cases

The simulation platform is envisioned as a system in which several simulation modules can be implemented, and that can support input and output interfaces for programming the simulation and interpreting results. The principal goal is to help testing and developing software in the C-AD control systems.

One of the purposes of simulation mode is to improve robustness of ADO codes by running through test data to improve software dependability. Some general testing guidelines can be followed. For parameters of numerical type, stress tests can be imposed, such as violating range boundaries or iterating every valid value for a given type of numerical parameter. For string type parameters, stress tests can be assignment with strings of huge size or zero size. For both types, tests of assigning inconsistent types of data can be performed. Some other special test scenarios are summarized as follows.

After upgrade of software, simulation mode helps developers to verify whether the new version of software works in a desired way by the following testing scheme. Simulation is performed with old logged input data, and new output is compared with old logged output data. Conclusions can be drawn based on whether the two outputs differ in an expected way. There are use cases where the aim is to make sure ADOs are ready to connect with some specific hardware when the hardware is not available, either because new hardware has not been deployed in the system or they are in a maintenance stage and cannot be accessed within a certain time. Test cases where control parameterization method is expected are also applicable, where only a part of parameters are of interests. In such situations, parameters of interest are varied in a random or controlled fashion while the rest are applied with fixed test data.

FUTURE WORK

Our focus thus far has been on defining the overall structure of a useful control system simulation environment. The system we have described does that within the framework of the existing, real control system. One future goal is to construct a version of the system that can be completely self-contained. This would allow working in an environment in which all aspects of the system might be controlled, without impact on the actual running control system. Another goal is to make the simulations more realistic. To do this means more work must focus on the hardware interfaces and developing modules that mimic the actual hardware. The system is also not really ready for general release to developers. The user interface remains primitive and more automated analysis needs to be added. One incomplete piece is the construction of a special client that monitors both the inputs to an ADO and the output. We imagine this special client to become the main user interface developers could use to construct and perform testing suites and even develop regression testing groups.

CONCLUSIONS

In this paper, we present a way to assist in developing and testing software in the C-AD control systems. A new simulation architecture is proposed which mainly aims to improve ADO codes reliability. Key use cases of the testing platform are listed and analyzed. The next development stage is described.

REFERENCES

[1] K. A. Brown, "C-AD Controls Systems various notes on history, architecture, and modern systems", in *Collider-Accelerator Department Documentation - Notes*, 2012.

[2] "Relativistic Heavy Ion Collider", `https://www.bnl.gov/rhic`

[3] T. S. Clifford, D. S. Barton and B. R. Oerter, "RHIC Control System", in *Collider-Accelerator Department Documentation*, 1997.

[4] T. S. Clifford, D. S. Barton and B. R. Oerter, "The Relativistic Heavy Ion Collider Control System", in *Collider-Accelerator Department Documentation*, 1997.

[5] D. S. Barton *et al.*, "RHIC control system", in *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, March 2003, vol. 499, issues 2-3, pp. 356-371.

[6] J.F. Skelly, J.T. Morris, "Design, Evolution and Impact of the AGS/RHIC Control System", in *Collider-Accelerator Department Documentation*.

[7] "Control System Tour: Interface to Accelerator Equipment", in *Collider-Accelerator Department Documentation*, 2013.

[8] J.M.Brennan *et al.*, "RF Beam Control System for the Brookhaven Relativistic Heavy Ion Collider, RHIC", in *Collider-Accelerator Department Documentation*, 1998.

[9] R. H. Olsen, "ADOGEN and .rad files", in *Collider-Accelerator Department Documentation*, 2016.

[10] R. H. Olsen, "Controls Name Server", in *Collider-Accelerator Department Documentation*, 2001.

[11] S. Sathe, "notifServer How To Guide", in *Collider-Accelerator Department Documentation*, 1996.

[12] S. Nemesure, "ADO Developers Guide To RHIC Alarms", in *Collider-Accelerator Department Documentation*, 1998.

[13] J. T. Morris, "AGS Control System Guide to Alarm Management", in *Collider-Accelerator Department Documentation*.

[14] T. D'Ottavio, "Controls Logging System", in *Collider-Accelerator Department Documentation*, 2011.

[15] M. Borland, "A Self-Describing File Protocol for Simulation Integration and Shared Postprocessors", in *Proceedings Particle Accelerator Conference*, 1995.

[16] W3C, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", `https://www.w3.org/TR/REC-xml/`